Ricardo Luís Correia Gonçalves Rocha

Degree in Computer Science

Discreet - Pub/Sub for Edge Systems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Sérgio Duarte, Professor Auxiliar,
Universidade Nova de Lisboa

Examination Committee

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2019

Discreet - Pub/Sub for Edge Systems

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

This work was partially supported by FCT/MCTES, through SAMOA project (PTDC/CCI-INF/32662/2017, Lisboa-01-0145-FEDER-032662) and the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FCT grant UID/CEC/04516/2013).

Abstract

The number of devices connected to the Internet has been growing exponentially over the last few years. Today, the amount of information available to users has reached a point that makes it impossible to consume it all, showing that we need better ways to filter what kind of information is sent our way. At the same time, while users are online and access all this information, their actions are also being collected, scrutinized and commercialized with little regard for privacy.

This thesis addresses those issues in the context of a decentralized Publish/Subscribe solution for edge systems. Working at the edge of the Internet aims to prevent centralized control from a single entity and lessen the chance of abuse. Our goal was to devise a solution that achieves efficient message delivery, with good load-balancing properties, without revealing its participants subscription interests to preserve user privacy.

Our solution uses cryptography and probabilistic data sets as a way to obfuscate event topics and user subscriptions. We modeled a cooperative solution, where publisher and subscriber nodes work in concert to route events among themselves, by leveraging a one-hop structured overlay. By using an experimental evaluation, we attest the scalability and general performance of the proposed algorithms, including latency, false negative and false positive rates, and other useful metrics.

Keywords: Publish/subscribe, Structured overlays, Privacy aware event filtering

Resumo

O número de aparelhos ligados a Internet têm vindo a crescer exponencialmente ao longo dos últimos anos. Hoje em dia, a quantidade de informação que os utilizadores têm disponível, chegou a um ponto que torna impossível o seu total consumo. Isto leva a que seja necessário encontrarmos melhores formas de filtrar a informação que recebemos. Ao mesmo tempo, as ações do utilizadores estão a ser recolhidas, examinadas e comercializadas, sem qualquer respeito pela privacidade.

Esta tese trata destes assuntos no contexto de um sistema Publish/Subscribe descentralizado, para sistemas na periferia. O objectivo de operar na preferia da Internet está em prevenir o controlo centralizado por uma única entidade e diminuir a oportunidade para abusos. O nosso objectivo foi conceber uma solução que realiza entrega de mensagens eficientemente, com boas propriedades na distribuição de carga e sem revelar on interesses dos participantes, de forma a preservar a sua privacidade.

A nossa solução usa criptografia e estruturas de dados probabilísticas, como uma forma de ofuscar os tópicos dos eventos e as subscrições dos utilizadores. Modelamos o sistema com o objectivo de ser uma solução cooperativa, onde ambos os tipos de nós Editores e Assinantes trabalham em concertadamente para encaminhar eventos entre eles, ao fazerem uso de uma estrutura de rede sobreposta com um salto. Fazendo uma avaliação experimental testámos a escalabilidade e o desempenho geral dos algoritmos propostos, incluindo a latência, falsos negativos, falsos positivos e outras métricas úteis.

Palavras-chave: Paradigma Editor/Assinante, Rede sobreposta estruturada, Filtragem de eventos com consciência em privacidade

Contents

List of Figures

List of Tables

Acronyms

FPP False Positive Percentage.

Pub/Sub Publish/Subscribe Paradigm.

RS Reed-Solomon.

Chapter 1

Introduction

In today's world, information is stored everywhere, from massive centralized data centers, to the edge of the network, distributed among peers across the globe. This reality is not the end result of a process, it is still very likely the beginning. In fact, data shows that by the end of 2017 there were 20 billion devices connected to the Internet and this number is forecast to grow up to 30 billion by 2020 [41]. While not all of this information is intended for human consumption, our exposure is bound to increase in coming years. A published study indicates that in 2012 we were already being bombarded with the equivalent of 174 newspapers of data a day [25]. It is easy to conclude that sooner rather than later, we will need increasingly better ways to filter what kind of information is sent our way. With so much information, our focus must be on what is important and avoid getting distracted in this overwhelming amount of information. We believe this will be vital for end-users but will be increasingly important for applications, as well.

Not only are we exposed to outrageous amounts of data, our actions while accessing this information are being collected, scrutinized and commercialized. The seriousness of the security and privacy concerns that this entails cannot be overlooked and it is now getting the attention of governments and authorities. An example can be found in the General Data Protection Regulation, known as GDPR [17], a policy passed by the European Union that aims to improve privacy laws, protect EU citizens data privacy and to reshape the way organizations approach data privacy [18].

Many of these issues arise from the fact that the very entities that store the data in data centers are those that strive to monetize our interaction with this data. User data has become a commodity by the gold rush mentality brought about by the recent big data and analytics movements. While user data can be used for commercial purposes, to present the user with products he may have interest in, more concerning is the use of user data to influence their opinion. A clear example of this abuse can found on the 2016 U.S.

elections, which Cambridge Analitica is accused of having influenced [40]. Moreover, it has become evident that many industry players are stockpiling data without the proper safeguards. As a result, when trusting them with our personal information we are in fact at risk of exposing secrets that we deemed secure. Personal information and account credentials improperly stored among this data are now the target of illicit activities.

If edge computing takes hold, instead of having all data and computation in centralized data centers, scattering both to the edge will help avoid or mitigate the issues presented. Performing more computing at the edge may be a way to diminish data exposition to centralized third parties. Namely, applications that have relied on centralized servers, may be replaced with others adopting more decentralized and collaborative models and architectures, especially in what concerns sensitive data storage and dissemination.

1.1 Overview

The focus of this dissertation was to study and research ways to disseminate information, in a decentralized manner, while taking into account the individual interests of each user without exposing those interests to others. To this end, we developed a publish/subscribe system that allows users to classify the information they desire to receive in the form of a set of subscribed topics. The system strives to deliver all information relevant to those topics and discard everything else. To address privacy concerns, the challenge was to allow efficient dissemination without exposing the actual list of topics each user has subscribed. This is a problem because, in order to discard information that does not match the user's desires, each piece of information needs to be evaluated against the list of subscribed topics. If this task is performed out of the user control, for example, in third-party machines, private information could leak. To tackle this problem, we studied ways of representing the user's interests that allow for efficient filtering but do not allow other parties to determine the actual topics being matched. Therefore, fully decentralized cooperative dissemination algorithms are possible without the risk of exposing each user's subscription patterns and interests. With our general goal stated, the remainder of this chapter will provide a quick overview of the problems, methodologies and uses on the type of systems our research focused on.

We will start with the publish/subscribe architecture pattern, where message senders are called publishers and the receivers are called subscribers. As the name suggests, subscribers can express interest in particular messages by subscribing to them, independently from where they originate.

There are several categories of publish/subscribe systems, such as topic-based, content-based or channel-based. In the latter, messages, also called events or notifications, are organized in distinct separate flows. In topic-based systems, publishers are responsible for defining the topics or classes of messages to which subscribers can subscribe by tagging each message with the topics it belongs to. In the content-based approach, the subscriber is the one responsible for classifying the messages, typically using some kind

of subscription language that operates over the contents of the message. This way it will only receive messages that match the constraints defined by itself.

These messages can be sent to the subscriber using an intermediary message broker responsible for doing the message filtering. The intermediary message broker allows for a loose coupling of the system's components, this is, the components have little or no knowledge about the other separate system components. This not only represents a location decoupling but a space decoupling as well. Thus, enabling publisher and subscribers to work with no space and time affiliation. A widely used centralized pub/sub system is Apache Kafka [44]. It's also possible for a system to exist without the message broker, to do so the publishers and subscribers need to share metadata, namely location and interests about each other. We intend to focus our approach in these specific types of systems, which rely on decentralized solutions. We will provide a more specific overview on how decentralized pub/sub systems are built later on.

Fully decentralized solutions are often implemented at the periphery of the Internet. They target environments with very large amounts of nodes, wide-spread geographical distribution, which results in weaker and slower links between them. Furthermore, the participating nodes are heterogeneous and predominantly connected by wireless links, resulting in an assorted mix of processing power and a high tendency to join and leave the network constantly. In fact, a well know problem in these systems is the continuous process of node arrival and departure, known as churn. The sum of these characteristics results in overall more complex protocols.

Message forwarding in these periphery systems requires participant nodes to organize a decentralized solution, communicating with each other without the need of a centralized server. This is known as a peer-to-peer overlay network. This was an area of intense research in the early 2000's, as it became popularized with file sharing applications, such as Napster and BitTorrent[8]. Research in this area has since regained popularity, with the growing interest and traction of the Internet of Things, Fog Computing and Mobile Edge Computing. These environments strive to relocate processing power from the center of the internet where core computers and links have great capacities, to the edge of the network. At the edge, devices are not as powerful, links cannot withstand such bandwidths as the core, so systems targeting this environment face issues with performance, latency, scalability and even security[14].

Today, edge computing is a term that covers a variety of environments, ranging from small wireless devices (IoT) to (virtual) machines located at small data centers hosted by ISPs. The focus of our research is towards the latter. Namely, we are interested with the kinds of deployments that will be possible with upcoming 5G standard, where the total amount of data the network can serve will increase by 1000 times from the 4G to 5G [3] and will allow some form of application-level computation to be performed at base towers. This standard will use several key technologies to accomplish this, but we are interested mostly in the *densification* and offloading features, which is done by making base station ranges smaller. In fact, networks are evolving to include stations

with ranges only up to 100m. This strategy has proven to be an efficient way to increase network capacity, but also significantly increases the number of base stations. As such, the number of participant nodes in our solution overlay network will approach those found on classic volative P2P networks. However, in contrast to those, our focus was on more reliable networks with stronger connections, which will not fail or disconnect frequently, and will have more processing power.

Decentralized P2P pub/sub systems can be modelled as two distinct message dissemination layers. A membership layer is responsible for organizing the nodes in the system, and a routing layer where pub/sub events are routed using the information and facilities provided by the former. The performance of the resulting solution is inherently tied to the topology of the overlay network used by each of these two layers. In particular, if participant nodes that are neighbours at the level of the membership layer share the same or similar subscription interests then it is more likely the routing layer will be able to minimize the need to use intermediate nodes to forward events that they themselves do have interest in.

On the other hand, in [24] and [23], it has been shown that is feasible to create membership layers that encompass all or a significant fraction of the participant nodes in a P2P system. Interestingly, the cost of such approaches is tied to the churn of the system and not to its actual size. As such, we applied this design to our solution, as a way to enable each participant node to have a very complete view of the subscription interests of the other nodes in the system. As a result, at the routing layer, each node is able to locally determine which nodes are interested in a given event and select them as the best candidates to perform the next routing step. As such, in a sizable system, there will be multiple options when assembling the event dissemination tree.

In tree-based pub/sub solutions, the role of the centralized broker is shared among many nodes. This means these will need to match events against participant interests and also perform work by forwarding messages to other nodes in the system. However, terminal nodes will only receive events. This leaves the system unbalanced as some nodes always perform work, and others do not. Changing the dissemination tree helps to rotate the role of participant nodes and leads to a better load-balanced solution.

Since we exploited a full membership layer, our solution is to able to change the dissemination tree often. Moreover, it is able to generate a dedicated random tree, on-demand, for each event, which encompasses only those nodes whose interests match that particular event. Not only this is fairer but, from a security point of view, it is useful because it also avoids fixed communication patterns.

To address the security issues of the build system, we had to take special concerns with the representation of nodes interests or subscriptions. Since matching is performed in a decentralized way, every node needs to be able to match an incoming message to other participant's interests. To retain privacy, we want to be able to perform this operation without revealing the actual topics each node is interested in. To achieve this, we explored probabilistic data structures to model node subscriptions. In particular, we studied in

probabilistic sets, e.g, *bloom filters* and *cuckoo filters*, which allow querying for the presence of a given element in the set, but do not support listing the actual elements inserted into the set. These sets map items (in our case, topics) to some internal bits within the data-structure, which is only enough to support the query operation with high certainty. This mapping is achieved using a family of hashing functions that take the element to be inserted as its main parameter. To improve secrecy, we further concealed the topics subscribed by a given node by adding a cryptographic key parameter to the hashing process used in the aforementioned probabilistic data structures. This allowed to further segregate and conceal nodes interests according to a mutual trust criteria. To interact, nodes need to share interests and use the same key to encode and decode topics (besides the event contents), otherwise they would operate separately, as if their interests are completely unrelated. This allows the system to support public and private topics at the same time, but only delivery private topics to the nodes that will be able to decode them.

Furthermore, there needs to be special attention to nodes that have very narrow interests. These nodes might stand out from the rest of the network to an observer. To mitigate that issue, a technique is to add spurious subscription topics to the nodes affected by this condition. This represents more work for them, as the augmented subscription would deliver unrequested events (false positives) and incur in more work for said nodes. Nevertheless, this is an interesting trade-off for better privacy.

The environment on where we are developing our system implies there is good connectivity and bandwidth among the nodes. Although, we propose our solution for two different scenarios. In the first scenario, the system does not suffer from churn, as a classic P2P system. An occasional node joining or leaving might happen but it should not significantly affect the performance of the system. However, in this scenario, the participant nodes will act as servers to the actual subscriber nodes, for example, mobile devices. As a result, we had to consider the impact of having to update node subscriptions as the end clients come and go. This is akin to churn, in some ways, except that when the combined interests become narrower it is not necessary to update the server subscription if a small amount of false positives is acceptable. In a second scenario, participant nodes are the final destination for exchanged events. This scenario shall be characterized by having a large number of nodes, with modest session durations. As such, participant nodes are not expected to change their interest during their limited sessions and filter updated shall be rare.

Finally, it is important to measure the performance of the system we developed. An essential criteria in the kind of systems is the number of false positives and false negatives, which can be described as:

- **False positives:** A node receives a message when it should not. As in, message is delivered to a node which did not subscribe to a certain topic. This behavior is undesirable as it leads to inefficiencies and so needs to be minimized.

- **False negatives:** A node does not receive a message when it should. This is, a message is not delivered to a node that subscribed to a certain topic. This type of errors must not happen in a correct system.

Overall, the most important metrics evaluated in our system were the number of false positives, number of false negatives, the effectiveness of the load balance solution and cost of keeping the membership layer. While the environment where our system was developed does not have as many nodes as extreme edge environments, it still is characterized by thousands of nodes. Therefore, we had to resort to simulation in order to conduct our experimental evaluation.

1.1.1 Contributions

The developed work provided the following contributions:

- The model for a decentralized Publish/Subscribe aimed for the edge of the main Internet. The system distributed event routing work in a fair manner, without compromising the privacy of the participants, keeping the topics and event payloads concealed from the rest of the system.
- A real-world prototype of the Publish/Subscribe model described.
- A valid and extensive experimental evaluation, that attested the goals of our system through the use of the simulated environment and real-world environment.

1.1.2 Document Structure

The remainder of this document is structured as follows. In Chapter 2, we introduce and analyze the major areas of existing work related to this thesis. The chapter is divided into subsections that cover the following areas, probabilistic data structures, error correction codes, peer-to-peer overlay protocols and decentralized publish/subscribe systems. At the end of each section, we discuss the advantages and disadvantages of each topic and how it relates to the proposed work. Chapter 3 provides a detailed explanation of the proposed system solution. In Chapter 4 explains how the proposed solution was implemented. The chapter clarifies the used structures, libraries, and some other implementation details. Chapter 5 presents a experimental evaluation of the solution. It demonstrates the correctness of some system proprieties and its performance according to the previously described metrics. Finally, in Chapter 6 we draw a conclusion about the work and possible future developments.

Chapter 2

Related Work

This chapter will be structured in the following way. In section 2.1, we will talk about probabilistic sets and how useful they are to our approach. Section 2.2 explains error correction strategies and how they might be useful for our work. Section 2.3 discusses peer-to-peer overlay algorithms, and gives an insight to some solutions for unstructured and structured overlays. Section 2.4 will discuss topic-based and routing-based routing solutions.

2.1 Probabilistic Sets

Probabilistic Sets are a type of data structure. The main goal of these structures is to minimize the amount of memory used on queries, with the drawback of possible false positives. This can be achieved by converting a set of elements through a hash process. The result of each element consists of positions on a set of bits. To test for the presence of an element, this kind of data structure digests the item through the hashing process and checks if the resulting positions are present on the set. As different elements can be mapped to the same positions on the set, it is possible to have false positives. False negatives are not possible. In other words, it allows testing if an element is "possibly in the set", or "not at all in the set". In this section, we study two different approaches.

2.1.1 Bloom Filter

A Bloom filter is a kind of probabilistic set implemented as a bit array where all bits are initially set to zero [9]. Different hash functions map an element to one of the positions on the array. It is important to note that the number of hash functions is a parameter of the system and that should be much smaller than the size of the array. For an element to be added to the filter it has to be digested through all hash functions. The resulting

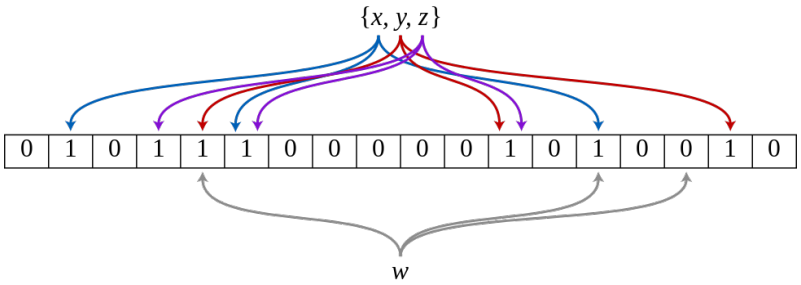


Figure 2.1: Example of a bloom filter [47]

positions returned by the hash functions are then set to 1. Figure 2.1 shows examples of queries done to the structure. Elements x,y,z are digested through the hash functions and the corresponding positions set to 1. Element **w** is digested through the hash function as a get query. One of the positions is set to 0, meaning element **w** is not present in the filter.

False positives can occur when two different elements intersect the same positions on the array, or diverse elements fill the array in such a way that matches another element. The rate of false positives can be decreased by increasing the number of hash transforms, up to a point, or increasing the array size.

In traditional Bloom filters, it is not possible to delete an entry without degrading the performance of the filter. Counting Bloom filters [20] uses an array of counters instead of an array of bits, in order to allow deletions. As such, this strategy requires more space than traditional Bloom filters.

2.1.2 Cuckoo Filter

Cuckoo hashing is a type of collision handling used in hash based data structures [19]. Collision handling is the procedure done by the hashing table when a key is hashed to a bucket that already contains a value. Cuckoo hashing is done by hashing a key with two different hash functions, meaning the value can be assigned to one of two buckets. If the first one is empty, then the value is placed in that bucket. Otherwise, it is placed in the second bucket. In case the second bucket is already filled, the older value is replaced, resulting in an unassigned (key,value) pair. The algorithm tries to place the pair in one of the buckets, using the two hash functions process, and repeats it until all pairs are assigned. However, this process can result in cycles. Cycles can be broken by rebuilding the hash table either by making it bigger or changing hash functions. Although this strategy has good insertion times, cycles can make it drop significantly as the filter reaches its maximum capacity.

The Cuckoo filter is a variant of the cuckoo hashing that stores only fingerprints. That is, a bit string derived from the item using a hash function. Fingerprint sizes are determined by the desired false positive rate, meaning smaller rates requires longer fingerprints to allow for fewer false positives. Comparatively, Cuckoo filters can be more efficient and use less space than Bloom filters when the false positive rate is kept under3%.

This filter can extend the amount of data that can be stored by allowing buckets to hold multiple items, leading to a table space that can be filled to 95% with high probability. Cuckoo filters also support entry deletion without affecting their performance. Moreover, a variant denominated 2-3 Cuckoo filters [16] allows bitwise comparisons on cuckoo filter, finding the position that has a common fingerprint.

2.1.3 Discussion

Probabilistic data structures offer a good solution for minimizing the amount of space needed to store a set of elements. Although, this comes with a small penalty of having false positives. Keeping all topics in the plain can be an unnecessary space overhead for each node in the system. For this reason, the studied structures were selected to minimize the set length that contains subscription topics, at each node.

Cuckoo filters can have better space efficiency and performance when compared to Bloom filters, as proved in [19]. Although, as the table reaches its maximum capacity and it starts to recursively recalculate positions on insertions, a time overhead is introduced in the system. In our system, most of the operations will be queries, done during the routing process when nodes check filters in order to forward messages. We measure the performance of both filters, in a benchmark experiment I, and concluded that the benefits provided by Cuckoo filters not enough compared to the benefits of using Bloom filters. Bloom filters provide a simpler solution that allows for quicker filter queries.

Collisions or false positives represent unbalanced work to the system we envision, as nodes that do not subscribe to a topic will receive those events. Furthermore, this forces an additional unnecessary computation step, that also introduces latency to the system. So in order to avoid those the chosen filter should minimize false positives.

Both data structures convert information through hash digests. Hash functions are characterized by being deterministic and irreversible. The normal use case of hash functions in cryptography is for authenticity and integrity and not for confidentially.

In this case, the use of the hash functions does obfuscate topics in a certain manner. Since the hash functions are deterministic a topic would occupy the same positions on different sets of bits. If two nodes compared their set of topics, they would find out if they subscribe to some of the same topics. At the same time, they would not know what other topics each other subscribed to. Although, hash functions do not provide enough privacy and must be complemented with a cryptographic key, shared between participant nodes. Furthermore, the process does not cover the message payload, so there still is a need to encrypt the payload using secure cryptographic methods.

2.2 Error correction

Communication between the system's participants requires reliable data transfers channels. This can be achieved by implementing reliable communication protocols such as

TCP. Although, unexpected system behavior might cause missing messages. Techniques such as ARQ (Automatic Repeat Request) allow for missing packets to be retransmitted upon a timeout of an explicit request. The problem with ARQ is that it might be inefficient when dealing with uncorrelated losses, at different groups of receivers [35], like subscribers with different interests.

For those cases Forward Error Correction (FEC) techniques are possible. FEC is a technique used for controlling errors in data transmission over unreliable or noisy communication channels. The technique is based on error correction and correction codes. The message sender prevents losses by transmitting some amount of redundant information. This allows for the receiver to reconstruct the missing data, without any further communication, which reduces the time need to recover missing packets. Furthermore, the technique simplifies the system since a mechanism to explicitly recover messages becomes unnecessary. FEC comes at a cost of higher bandwidth in the system, as messages have to be sent with the extra redundant information.

Error correction codes can be explained in two main categories, block codes, and convolutional codes [49]. Block codes work on fixed-size blocks of bits or symbols. Classic block codes are Reed-Solomon, Hamming codes and BCH codes. Convolutional codes work on streams of bits of symbols of arbitrary length. The most common algorithm is the Viterbi algorithm. It allows optimal decoding efficiency while increasing the length of the convolutional code, at the expense of exponentially increasing complexity. Though to their seeming complexity, Convolutional codes were quickly discarded in favor of Block codes, which are better suited for the type of message in our environment.

2.2.1 Reed Solomon

Within the error correction types, Reed-Solomon is a widely used technique [50]. It is used in several multimedia technologies, like CD's, DVD's and in storage system such as RAID 6. It is also used in environments where total re-transmission of a message is expensive, like satellite communications.

Reed-Solomon treats a block of data as a field of elements called symbols. The mechanism is able to detect errors in the symbols and correct them. It transforms a initial message of k symbols into a longer message of n symbols. This process adds check symbols to the data, allowing the mechanism to detect wrong symbols. Therefore, it can recover the initial message from a subset of the n symbols. Further understanding of the algorithm requires a heavy mathematical analysis, which is out of scope for our work.

The approach used on RAID storage systems divides the initial message by blocks. The parity information, i.e., the information to rebuild the initial message, is distributed between the blocks. Parity distribution ensures that the initial message can be rebuilt in the presence of a faulty block. In the case of RAID 6, two blocks can fail and the message will still be reconstructed. Block division presents similarities with the needs of our system. A critical message could be divided into blocks and each block broadcasted

through the system. Receiving nodes can then reconstruct the message, even in the presence of failures.

2.3 Overlay Networks

An overlay network is a type of network that can be built on top of another network. It allows the system to create a virtual network that is independent of the underlying network. As the overlay is independent, it can assume any topology that might be required. Since these networks decouple themselves from the physical ones, it can mean a worse message delivery performance. This can happen as nodes that directly connect on the overlay, can be separated by several physical paths. Examples of overlays can be found on peer-to-peer systems.

We studied peer-to-peer overlay networks, as these have characteristics in common with our environment. In peer-to-peer overlays, communication is established without a centralized server, forcing participant nodes to collaborate in organizing the overlay network. These solutions often target environments with very large numbers of nodes, wide-spread geographical distribution, heterogeneous nodes and often wireless or weakly connected nodes. Weak connections tend to cause frequent joins and leaves on the system, resulting in a phenomenon called churn.

Churn can create instability in the system leading to, failed lookup requests, inconsistent results or consistent results but with dramatic increase in latency times [34]. When trying to solve churn problems, side effects need to be taken into account. For example: due to network congestion a node can experience timeouts, leading to the activation of a reaction mechanism, which leads to more congestion on the network. Furthermore, timeouts need to be chosen carefully. Timeouts too short can lead to additional bandwidth waste and too long can result in wasted time.

Even though the environment our work is targeting, is not mainly characterized by churn, peer-to-peer overlay networks have evolved to deal with that problem. Structured Overlays and Unstructured Overlays are the two major solutions. In the general case, both have to organize themselves without the need for a central coordination server.

The remaining of this selection is dedicated to studying overlays, followed by a discussion at the end of each category.

2.3.1 Structured Overlays

Structured overlays organize themselves in a specific topology, defined in terms of node identifiers or keys. A specific topology means the node graph is constrained. What makes an overlay structured, however, is that the overlay nodes are not free to connect to any other node. In structured overlays, node connections are established based on the node identifiers. Together, these two characteristics are the reason structured overlays allow for

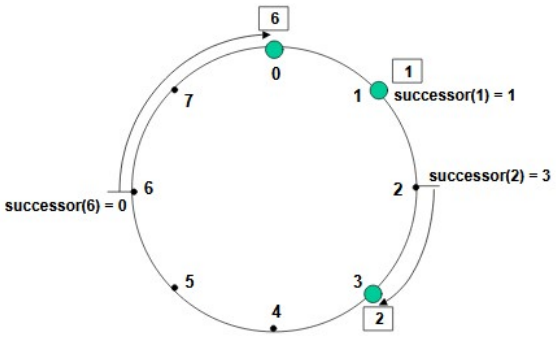


Figure 2.2: Chord key ring distribution [42]

efficient routing and data discovery [12], just as an ordered binary tree allows of efficient lookups whereas a simple binary tree does not.

In general, in structured overlays each node is associated with a key or identifier and has a neighborhood definition with nodes whose keys are close to them, according to some metric. When used for data storage, data items are also associated with keys and mapped to responsible nodes. By knowing the structure of the overlay nodes, it is possible to perform exact match queries, in an efficient manner. Given this property, structured overlays are widely used as distributed hash tables (DHT).

A loose definition sometimes exists between semi-structured and structured overlays. The former allowing nodes more freedom to find partners in the overlay network. For simplicity, we are going to assume that protocols that organize the overlay based on keys and some notion of a topology neighborhood, as simply structured overlays.

Chord

Chord is a decentralized **DHT** protocol that automatically adapts to hosts leaving and joining with little key movement[42]. It provides load balance for keys which is done by using consistent hashing, spreading keys evenly over the nodes. It addresses availability by adjusting its internal tables, even in the presence of failures. This way, the node responsible for a key can always be found.

Key assignment works in the following way. The identifiers orientation can be seen as a circle, in which the last identifier will be succeeded by the first one in the circle 2.2. A node is called a successor of a certain key if its identifier is the next in the identifiers space. In other words, every key between a node **n** and the next node **n+1**, belongs to the node **n+1**. When a node leaves the system, all of his keys are assigned to the next node, its successor. So a node should have $^1/_N$ of the keys every time the network is stable. This also means load should be distributed in a fair manner.

Nodes keep a local table in which they store information about a small number of nodes close to themselves. The table is used for key and node lookup. If the information is not enough to locate a successor of a key, it finds another node closer to the key than

themselves. This measure is designed to avoid space overhead, as each node only has a small amount of the routing information.

Path length is one advantage of Chord since the mean path length is $\frac{1}{2}logN$. Other advantages are, it is simple to implement, has a proven correctness, scales well, and can correctly recover from simultaneous node failures. Furthermore, when nodes fail, the amount of failed lookups on some query is directly proportional to the number of nodes fails. This is expected since those nodes were responsible for those keys. On the downside, Chord has no specific mechanism to heal partitioned rings. Moreover, Chord nodes might need to ask others for a key several times before realizing where it is, leading to an increased lookup latency on the system.

Kademlia

Kademlia is a peer-to-peer lookup protocol based on a distributed hash table (DHT) that uses keys with 160 bits[31]. It distinguishes itself from other protocols as it uses a XOR metric system to calculate the distance between points in the key space. It simultaneously offers several features, such as the minimal number of configuration messages, automatic spread of configuration information, parallel and asynchronous queries. Furthermore, the protocol was build using a single routing algorithm. This is opposed to other protocols that often use one algorithm for the first mass dissemination, and another when the message is on the network periphery.

Kademlia treats nodes as leaves in a binary tree. Thus, leading to each node position being determined by the shortest unique prefix of its ID. Keys are assigned to nodes based on a distance between two node identifiers, which is given as their bitwise exclusive or XOR. This is done by implementing a bucket system which is updated accordingly to least-seen policy. The protocol has a preference for older nodes on the bucket list. Since the longer a node has been up, the more likely it is to remain connected. This technique also helps prevent DOS attacks as older nodes will be kept, avoiding new potential harmful nodes.

As the XOR metric in unidirectional, it allows for key lookups on the same key to converge along the same path. Therefore, searches are likely to hit cached entries. Store operations are only kept in the system for an amount of time before they expire. Defining the expiration timeout must be done accordingly to the purpose of the system. In order to ensure persistence, entries must be re-published.

Beyond Kademlia

Original Kademlia proposed protocol suffers from some issues, namely finding with accuracy all k-closest nodes to a target id. It also had problems finding all nodes within the close region of the target id computing the same k-closest nodes [26].

In the original paper, a node may not know himself as one of the k-closest nodes to an id. In order to ensure a correct computation of the k-closest nodes, in any given situation,

a node is initially root of a subtree. A subtree is split when it has more than k nodes and merged if it has less than k nodes. Merges can happen after node crashes and are performed with another close region with up to **k** peers. This provides ordered views, giving a definition of closed region, which leads to correctly compute k-closest nodes. As a means to improve availability in the network, a node not only keeps **k** peers, but also **h** extra peers. Those extra peers help the system tolerate crashes. If the number of peers drops below **k** nodes + **h** (H is a system defined parameter), then a node will try to obtain more peers to maintain that sum.

Kademlia prioritizes the longest running nodes, what may lead to a dependency on those nodes. If eventually those nodes fail, the system could become fragile. Having a more precise region definition and equally reliable nodes, as the benefit of a more uniform distribution on links load.

Dual-Kad

Dual-Kad is another protocol based on Kademlia. It proposes the combination of a peer layer with a super-peer layer[46]. The peer layer is described as being responsible for data storage, whereas the super-peer layer takes charge of supervising the routing table. Furthermore, the model is able to process queries based on semantic logic. This shorts query routing paths and overall speeds the routing processes.

A peer contains indexes or data. It also can belong to more than one super-peer. On the other hand, super-peers play a role of logical supervisor, managing which query is forwarded to which peer or super-peer. Super-peers keep a cache zone on where its peers will cache query results, allowing faster fetching of data.

The difference between normal Kademlia and this solution is that in Dual-Kad a peer will first check its super-peer table, and if its target it is not there, then the query will be routed over like in Kademlia. Since querying among super-peers is faster than peers, it allows for a shorter and faster querying process. Furthermore, they allow for the use of semantic queries which Kademlia struggles with, as it had to perform the lookup several times.

Pastry

Pastry is completely decentralized, fault-resilient, scalable and self-organizing protocol, that performs application-level routing and object location on overlay networks [36]. One of the main features of the protocol is that it takes into account network locality. It uses the number of IP routing hops as a metric, with the goal of minimizing message travel.

Each Pastry node has a routing table, a neighborhood set and a leaf set. Leaf set keeps a set of numerically closest nodeIds, so given a key a node first checks if the key is covered by his leaf set. The routing table contains IP addresses of nodes closest, according to the proximity metric. The neighborhood set contains a list of closest nodes, that are mainly used to maintain locality properties.

If the leaf set does not cover the key, then the message is forwarded to a node that shares a common prefix with the key by at least one more digit. In case it does not exist in the routing table, then the message is forwarded to a node which is numerically closer to the key then the present node. The routing overlay can be seen as a mesh [37].

A new node enters the network knowing a contact node, which informs the numerically closest nodes of the new node about its entry. This message passes through several nodes, which send their tables to the new node. That information is used to build the new node tables.

According to the authors, "Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no sense of global direction."[36]. It may miss nearby nodes with a different prefix than the key since it routes primarily based on node id prefixes. Although, results show that it finds the closest node most of the time. Furthermore, the average number of routing hops is $\log_{2^k} N$ (where k is a system parameter). Results show that in presence of node failures Pastry can recover missing table entries, even without using its repair mechanism. However, it requires ring multicast search to avoid isolated overlay networks, which can appear over network anomalies.

Tapestry

Tapestry is a peer-to-peer overlay routing network that takes into account network distances. It provides decentralized object location and location-independent routing using only localized resources [51]. It works in a type of mesh routing, incrementing a common prefix at each hop, like Pastry, but with some differences on how they handle network locality and data object replication [37] [30].

Tapestry mapping leads to each root having a unique spanning tree for routing. Each node has a neighbor map with multiple levels. Each level contains links to nodes matching a prefix up to a digit position in the ID space. This method ensures that nodes will be reached in at most $\log_b N$ (with IDs of base b) logical hops. When the algorithm encounters an empty neighbor entry, it routes the event to a node with a close prefix on the routing table. This technique is called surrogate routing.

Furthermore, the protocol stores the location of all data object replicas. This measure increases semantic flexibility and allows the application to choose from a set of data object replicas based on some selection criteria. To ensure reliable routing in the presence of faulty links each node has backup links, each sharing the same prefix.

CAN

CAN is a completely self-organizing peer-to-peer protocol. It aims to accomplish scalability, robustness, and low-latency in a hash table operation fashion [33].

The protocol works around a cartesian coordinate space. Each node stores a logically divided zone, called a chunk, and information about a few adjacent chunks. This coordinate space can also be multi-dimensional. Therefore in a space with d-dimensions, a

node is considered a neighbor when their space overlaps along **d-1** dimensions. Using the space set, messages are forwarded to other nodes with coordinates closer to the zone that contains the key. A new node randomly chooses a point P in the space and sends a join request destined for P. Occupant node of that zone splits it in half, and assigns half to the new node. Zone splitting is done in a way that allows it to be re-merged later, should one of the nodes leave. Nodes send periodic messages to their neighbours containing: their zone coordinates, a list of their neighbors and their own zone coordinates.

Increasing the number of dimensions in coordinate space reduces path length and latency, at the expense of increasing routing table size. This measure leads to a node having more neighbors and improving routing fault tolerance. An additional feature allows to maintain multiple independent coordinate spaces, thus increasing availability. Furthermore, the protocol can organize itself using physical closer nodes. However, if that option is used, the coordinate space is no longer uniformly populated.

One Hop Lookup for Peer-to-Peer Overlays

One hop Lookup protocol maintains complete membership information at each node [24]. It proves that nodes can maintain a full membership with low communications costs. Thus, accomplishing fast routing at the expense of large routing tables.

Nodes are arranged in an identifier ring modulo of 2^{128}bits, similar to Chord. The ring is equally divided into slices, having each slice a leader. The leader is the successor of the midpoint of that slide identifier space. Each slice is also divided into equally sized intervals called units. Units leaders are elected in a similar fashion to slide leaders. When a node detects a change in the membership it notifies his slice leader. Slice leaders collect information for a period of time, before disseminating that information to all other slice leaders. It is important to note that this dissemination is not synchronized in order to avoid bursts messages that could cause network pikes. Slice leaders will also wait for a period of time before disseminating events to their unit leaders, whom will propagate messages the rest of the ordinary nodes. To help with scalability nodes who have a better connection are identified as super nodes while entering the system. This leads to a parallel ring of super nodes. Those can be used to select slice successors, as they perform more work than other nodes.

Imposing a structural division in the system, with defined dissemination trees helps to ensure no redundancy in communications. The timers described above help aggregate several events into one single message, reducing message overhead. Furthermore, the message cost on this system is proportional to the amount of churn. This means that in stable environments it has lower bandwidth requirements.

Two Hop Lookup for Peer-to-Peer Overlays

One-hop solution might require too much bandwidth for most of the nodes on very dynamic systems[23]. The same author proposed a two-hop lookup solution. It results

in less bandwidth and it is more scalable for systems that exhibit more churn, while still being faster than multi-hop lookups.

Node arrangement and membership changes are handled in the same way as the one hop solution. Every slice leader chooses k-1 groups of nodes containing nodes from its own slide. Each slide leader exports information about a group to exactly one other slice leader. This information is then passed to the members of that slice, which will choose the node closer to them from that group. It leads to every node having the closest node to him in every slice. Nodes still keep all information about nodes in his own slice. Table sizes are dependent on the size of **k**. For a system with 10^8 nodes the recommended value of k is 1500. Low bandwidth is also an important feature of the protocol which is demonstrated to be several times lower than the one hop solution.

2.3.2 Discussion

	Table Size	Routing Cost
Chord	$\mathcal{O}(\log n)$	$\mathcal{O}(1/2 \log n)$
Kademlia	$\mathcal{O}(k \log n)$	$\mathcal{O}(\log_2 n)$
Pastry	$\mathcal{O}(\log_{2^k} n * 2^{k-1})$	$\mathcal{O}(\log_{2^k} n)$
Tapestry	$\mathcal{O}(k \log_k n)$	$\mathcal{O}(\log_k n)$
CAN	$\mathcal{O}(2k)$	$\mathcal{O}(n^{1/k})$
One-Hop	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Two-Hop	$\mathcal{O}(k + n/k)$	$\mathcal{O}(2)$

Table 2.1: Table Size and Routing Costs for Structured Overlays. Variable **n** represents the number of nodes in the system. Variable **k** is a small parameter, that does not change with the size of the system.

Structured overlay protocols try to deliver the message in a few steps, directly to a specific node, without using flooding techniques. Chord [42] is one of the base structured protocols, organizing the overlay by doing key hashing. Kademlia [31] organizes its table using a XOR metric system to calculate the distance between points in its key space. Beyond Kademlia [26] proposes to solve original Kademlia problems by finding all k-closest nodes with accuracy. Pastry takes into account the number of IP routing hops as a metric to build its tables, and it so by doing prefix matches [36]. Tapestry [51] offers a similar solution to Pastry, doing location routing by matching prefixes at each step. CAN [33] offers a very different solution as it divided the key space in an n-dimensional space. One-Hop Lookup [24] offers a solution where it can route to any node in one-hop, with the trade-off of having larger routing tables. Two-Hop Lookup [23] has one more routing hop, with the benefit of smaller tables and less bandwidth, when compared to the One-Hop solution.

Table 2.1 shows the table size and routing cost for each structured overlay protocol. It is possible to observe that lower routing costs come from bigger tables. Nodes in our

target scenarios will have fairly good capacity, so we feel the trade-off on keeping a bigger routing table seems beneficial.

Moreover, we also introduce randomness in order to avoid pattern seeking. Patterns can be exploited to target users in our system. Keeping a more complete routing table could help a node randomly route each message via different nodes. Therefore, we argue that maximizing the routing table size will also help in this respect.

A common belief is that structured overlays are expensive to maintain. This is not a problem since a big part of routing costs in these types of solutions are associated with a node joining or leaving the network. Once again our deployment environment is expected to have low levels of churn, which should curb membership routing costs.

2.3.3 Unstructured Overlays

Unstructured overlays are formed by nodes that randomly establish connections among each other. They do not impose constraints on the network topology. This allows nodes to connect at any point, hence they can be neighbors to any other node on the network. It does not mean unstructured overlays strictly have to form random topologies. It does however mean, node organization is more loose, as opposed to structured overlays where keys define where and with whom a node can connect. As a result, these protocols tend to be very flexible and designed to resist high levels of node failure.

Since nodes are not organized in a specific manner, it is not possible to accurately find a specific node without probing large parts of the network. In fact, often these types of protocols rely on expensive message delivery mechanisms such as flooding. This means that in order to avoid infinite cycles, nodes need to cache a list of recently received messages. Whenever a node receives a message or query during a flood it evaluates it based on previous messages that it received, only forwarding new ones. This common forwarding technique is related to gossip message dissemination.

The remaining of this section overviews several unstructured protocols.

Scamp

Scamp is a simple decentralized peer-to-peer membership service based on gossip dissemination. Nodes keep a partial view of the system, which is capable of adapting accordingly with the size of the network[21].

When a node receives a subscription request from a new node, it forwards said message to all members of its own local view. Nodes that receive the join operation can add the new node to their views. This happens based on a probability, which depends on the size of their view. Furthermore. nodes create **c** additional copies of the subscription message and send them to random nodes of their local view. According to this metric, each view configures itself towards a size of $(c+1)\log n$, where n is the number of nodes in the system.

Scamp has a strong theoretical proof of its design. It manages to ensure gossip delivery with high probability and its robust to node and link failure. Results show that the network remains connected, with good levels of reliability. Although, these results are only shown to failures up to 50% of the nodes.

Cyclon

Cyclon is an unstructured membership management overlay for very large peer-to-peer networks[45]. Using gossip dissemination it aims to be highly scalable, robust, decentralized and inexpensive.

When a node joins the system it initially knows a contact node. The contact nodes starts **c** (cache sized) random walks, with the new node id on them. Nodes where those random walks end will replace one of their entries for the new node. Replaced entries are sent to the new node, who has to include them in his empty cache.

Cyclon uses a periodically shuffle mechanism where at each round a node P increases his neighbor's age. After that, it selects the oldest one to shuffle (node Q) and sets that neighbor age to 0. Node P sends to node Q a random subset of nodes in his cache. Node Q sends back a random subset of his own cache. P discards entries pointing to himself and includes the remaining entries in his cache slots. Nodes that do not respond to a shuffle are considered as failed and removed from the cache.

This shuffling technique does a better job than normal shuffling in respect to spreading out evenly across all nodes. It results in better in-degrees and a more up-to-date overlay. Furthermore, it has very small average path lengths and a clustering coefficient similar to random graphs values. Cyclon resists up to 80% of node failure without affecting its reliability, but it still manages to repair the overlay over those values. Even though it is a gossip dissemination protocol, it is also considered inexpensive in terms of bandwidth.

HyparView

HyparView is a membership protocol to support gossip-based broadcast[29]. As gossip in its basic form is not scalable, the protocol creates an overlay network using partial views. This strategy allows for a scalable message delivery system, that ensures high levels of reliability even in the presence of extreme node failure. It uses TCP as a failure detector, to provide fast healing properties, which is ideal for gossip protocols. Moreover, it allows for smaller out-degree fanouts, guaranteeing a more cost-effective gossip

The protocol works in the following way. Each node has two distinct views, a small active view, and a larger passive view. The active view purpose is to be used as an overlay for message dissemination. The passive view is a list of nodes that can be used to establish communication in case of failed active view members. When replacing a failed node from the active view with another node from the passive view, the connection can fail to the target replacement. In that case, that node gets removed and the process is repeated until a connection is established.

The passive view is maintained using a shuffling operation. The node that initiates the exchange will forward a list containing its id, a subset of nodes from its active view, and another subset from its passive. This list is propagated using a random walk strategy until its TTL expires. The node that receives the list will send its own list to the original sender, and both should integrate the elements to their passive views.

Through the use of deterministic node selection, for forwarding gossip messages and symmetric views, the protocol ensures 100% reliability, as long as the overlay remains connected. This way HyparView is capable of sustaining a high level of node failure while still maintaining high reliability. It accomplishes these features while still using a small fanout. Furthermore, healing time in the overlay is also quick, close to one or two rounds for node failures below 80%. Slow nodes could cause a blockage in the overlay because of the TPC flow control. This leads to having to consider slow nodes as failed ones and remove them from the active views. Otherwise, neighbours of that node could block and spread the problem in an epidemic manner.

PlumTree

PlumTree is a message dissemination protocol that uses tree-based broadcast primitives as its main message broadcasting property[28]. Normally tree-based protocols have lower message complexity but suffer in case of failures when compared to epidemic protocols. The protocol is designed for fast recovery and tree healing, in order to achieve low message complexity and high reliability at the same time. Moreover, it is important to highlight that the protocol was built on top of a peer sampling service, more specifically HyparView [29].

PlumTree relies on the use of TCP to ensure extra reliability and an additional source of failure detection. Message delivery is done using two different gossip strategies, eager push, and lazy push. On eager push nodes send the message payload to random nodes, while on lazy push they only send the message id. If a node has not seen the message id before it will perform a request for that message payload.

The links used for eager push are selected in such way that they build a broadcast tree. An important feature of this approach is that as long as failures are not detected, the set of random peers is the same in each gossip round. Since eager push over a broadcast tree is not enough to ensure message delivery in case of failures, the protocol recurs to the lazy push set. This is not only used as a way to recover missing messages, but also as a quick healing mechanism. Plumtree is optimized for a specific sender, although it can be used with multiple senders. To accomplish that, each sender should use a different instance of the protocol, or a single instance can be used by multiple senders.

In stable environments, the protocol can deliver messages to all nodes with zero redundant messages. Compared to HyparView it produces roughly the same amount of messages. Although, 75% of this are control messages which are much smaller than payload messages, causing less exhaustion in the network. It can recover quickly from

failures if the protocol is configured for a single sender. Even while facing failures the protocol can achieve 100% reliability and still maintain the number of redundant messages close to zero. As new nodes or links appear in the system, the spanning tree does not always evolve to accommodate new, better paths. This happens since the tree is produced based on the first broadcast message. Although, a solution is proposed based on changing the eager push link to a lazy push link with a lower hop count. The solution is based on a threshold value, that needs to be adjusted accordingly with the number of senders in order to avoid constant change.

2.3.4 Discussion

Scamp [21] offers a very basic solution based on local views. Cyclon [45] introduces the shuffling technique in order to keep the overlay more up to date and resist node failures. HyparView [29] evolves using two different partial views in order to resist massive node failures. These three protocols form random overlays based on local views, that are later used to disseminate message using flooding techniques. PlumTree [28] takes a different approach as it builds a tree-based broadcast primitive.

It is important to notice that the studied unstructured overlays were built with the main goal of being robust. As a trade-off, their routing tables are smaller. Finding rare items cannot be done efficiently since it requires flooding the entire system. It is also harder to control the rate of false positives due to the overlay characteristics, and delivering messages to nodes that are not subscribed should be avoided. PlumTree is an exception to this last case, as it uses dissemination trees. Nevertheless, PlumTree is impractical to implement in systems where all nodes can be senders. Nodes in our target scenarios are not expected to suffer much from instability, nor do they need to keep small tables. Therefore, this type of system was discarded early on, as it goes exactly opposite to our planned direction.

2.4 Publish/Subscribe Protocols

Publish/Subscribe, also known as Pub/Sub, is a message delivery architecture pattern. If subscribers have interest in a particular type of message, they can subscribe to it, independently from who will publish it. These systems can be divided into several categories, topic-based, content-based and channel-based, according to they way subscribers express their interests.

In topic-based systems, publishers are responsible for defining the topics or classes of messages to which subscribers can subscribe by marking each message with the topic(s) it belongs to. In the content-based approach, the subscriber is the one responsible for classifying the messages, typically using some kind of subscription language that operates over the contents of the message. This way it will only receive messages that match the

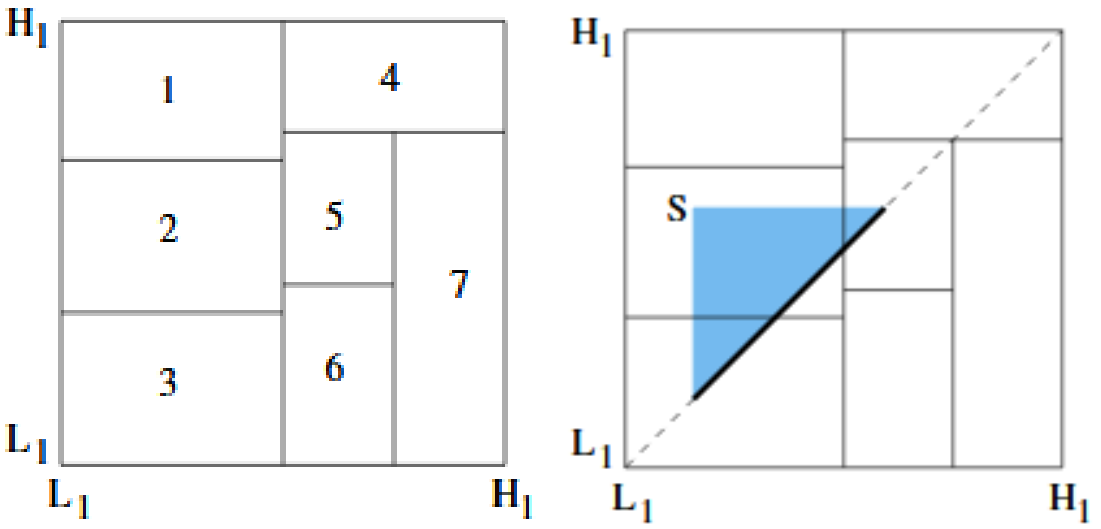
constraints defined by itself. In channel-based publish/subscribe systems, messages, also called events or notifications, are organized into distinct separate flows.

Pub/Sub systems can rely on centralized or decentralized solutions. In centralized solutions, published events are sent to a centralized message broker. The broker is responsible for filtering messages, sending them to the appropriate subscribers. This allows for a loose coupling of the system's components. In other words, the publisher and subscribers have little or no knowledge about each other. The decoupling also allows publishers and subscribers to operate at different periods of time. Hence, there is no space or time affiliation. Moreover, it distinguishes itself from typical message protocols since participants do not communicate directly with one another.

In decentralized Pub/Sub systems, there is no centralized message broker. This implies that in order to route messages, participants need to share metadata, namely location, and interests about each other. Decentralized systems can be modeled in two distinct message dissemination layers. A membership layer organizes the participant nodes in the system. A routing layer is responsible for routing pub/sub events using the information provided by the membership substrate. Some systems combine both layers, aiming to optimize for a particular aspect of that particular system.

The remaining of this section overviews different strategies for event routing in pub/-sub systems.

Meghdoot



(a) Division of the cartesian space (b) Region of events affecting a Subscription

Figure 2.3: Meghdoot dimensional organization [22]

Meghdoot is a content-based scalable pub/sub system, that uses a structured DHT to store and route subscriptions[22]. The system uses a well-known description of subscription, where the system itself can be described as a set of attributes. A subscription

is a conjunction of predicates of attributes, and an event a set of equalities over the attributes in the schema. An event is said to match a subscription if each predicate of the subscription is satisfied by the value of the corresponding attribute specified by the event.

Peers are organized in an n-dimensional DHT as described in CAN [33]. Taking into consideration CAN cartesian arrangement, an attribute **Ai** can have a domain with range [Li,Hi], as shown in figure 2.3a. This is, the predicates of a subscription specify ranges of interest over the attributes, and the ranges are represented by a point in the logical space. Subscriptions can be seen as a region on the Cartesian plane, shown in figure 2.3b. A new event in the system will be routed to a node in the region affected by the event and then to that node neighbors that subscribe to that event. The initial routing means that some nodes might have to do work for events they do not subscribe to. Reference points ensure that none of the affected neighbors are missed, and prevent back propagation of events. The way the events are propagated allows for a node to check if bottom right neighbors could have already delivered the message to the next neighbor. In that case, the node would not propagate the event to that neighbor. This means event propagation flows in a consistent and strict manner.

Meghdoot does not use a direct application of CAN since that would result in unbalanced load, caused both by subscriptions and events. In order to reduce subscription load, the protocol can divide a zone with a joining node. A zone can still be overloaded if it is in the propagation path for many events. Even after splitting that zone with another node, it will remain in the path of said events. Therefore the solution evolves creating alternative propagation paths to surrounding nodes.

Terpstra 2003

The protocol suggested by Terpstra, in 2003, is a Pub/Sub system that preserves the use of fully general filters. It can guarantee a small delivery time and evenly distributed load by using dissemination trees [43]. The protocol uses Rebeca algorithm for routing. Rebeca is an optimized spanning tree protocol for pub/sub networks. It also uses Chord DHT for its membership layer.

Each publisher is the root of its own spanning tree used to deliver events, avoiding routing cycles. Nodes that act as publishers not only publish their own events but can also publish events from other light or poorly connected nodes. A system invariant ensures that no early filters will reject a notification which would have been accepted later. Nodes in the topology keep a connection to the first node clockwise by a distance of $1/2^k$ or more along the circle. This a property from Chord that ensures routing tables of size $\mathcal{O}(\log n)$.

Event messages carry a range field that denotes the range a certain node is responsible for delivering the event to. The range is tested against the subscription filter and then divided into several sub-ranges. Each sub-range is forwarded to the first node that matches that filter inside the sub-range. This means a parent node will deliver the message to several child nodes, performing the routing on a tree-based fashion. The process is repeated

by all child nodes until no more targets are available. Thus, ensuring delivery's with a path length of log N.

TERA

TERA is a scalable pub/sub system [7]. It implements topic-based event routing architecture for very large scale peer-to-peer networks. Nodes organization is done in two different layers. A lower layer keeps a global overlay network connecting all the nodes. An upper layer is where topic subscribers are grouped into a dedicated overlay for that topic. Both layers use the epidemic broadcast protocol Cyclon.

Publisher nodes are able to publish events without subscribing to the topic they publish. Therefore, messages published by nodes are routed to an access point for said topic, through the lower level layer. Access points are nodes subscribed to a certain topic. Those, disseminate the received messages corresponding to their topic in their respective upper layer. Nodes periodically advertise the list of topics that they subscribe, to a random set of nodes in the lower overlay. Access points are kept on a limited sized table at each node. Each time a node receives an advertisement it will update the access point of that topic to the message sender. Thus, ensuring fresh access points and eliminating non-active topics. Furthermore, since the table is limited, nodes can search for other access points using a random walk.

The protocol as a built-in partition merging mechanism. Partitions can happen when two nodes try to subscribe to a topic with no access points at the same time.

Scribe

Scribe is a decentralized application level multicast infrastructure build on top of Pastry[11]. The protocol builds a multicast tree, formed by joining the Pastry routes from each group member to a meeting point associated with a group. Therefore, each tree ends up being associated with a different topic in the pub/sub system. Although the protocol is not directly described as being a pub/sub protocol, we assume the possibility to use it as such because of the way multicast groups are built. Similarly to Pastry, all decisions are based on local information.

Nodes can create groups, where each acts as a multicast source, the root of the multicast tree, or a group member. The meeting point or the root of the multicast tree are determined by selecting the node with the id closest to the group id, or by selecting the group creator. Nodes that forward messages in a group are called forwarders, and keep a table containing its children in the multicast tree. Forwarders may not be members of the group.

In real world situations not all nodes have equal capacity, so the protocol suggested an algorithm to remove bottlenecks from nodes. It allows nodes to offload their children to other nodes, reducing the amount of forwarding they have to do. Results show that

the protocol is not efficient with a large number of really small groups, as it can lead to high link stress.

Kyra

Kyra is a filter based pub/sub system for service networks. It aims to reduce the implementation cost of filter-based approaches, while still maintaining network efficiency [10]. Nodes in the system are not the end users, but intermediate servers. This solution is close to ours in the way that our nodes are not end users as well.

The protocol focuses on building multiple smaller filter-based routing networks so that routing at each layer has lower costs. Each node is incorporated in only a small number of subset networks. This is accomplished by moving subscriptions between servers, to improve content locality. Furthermore, each node only manages a small amount of information for each network in which it is inserted.

Kyra is divided into two layers. A bottom layer keeps nodes organized in groups based on their proximity, communicating with each other via unicast. Each node is assigned a non-overlapping zone in that group space. It becomes the proxy server for all subscription in that group, meaning original receivers will forward events to them. Partitions at each group are only visible inside that group.

An upper layer uses routing trees that establish connections between groups. Routing trees are assigned to non-overlapping content zones and used to route all events falling into that zone. Furthermore, each routing tree is an independent filter-based routing network. This means as an event enters the system, it will be forwarded to the proxy node in that group. The proxy node will then forward events using the respective routing tree, to nodes on other groups that cover the same subscription topic.

SpiderCast

SpiderCast is a scalable, churn-resistant pub/sub system. It uses partial views to reduce the average node degree while still guaranteeing that events are routed through nodes interested in that topic [13]. The protocol does not specify the membership substrate. It states that each node needs to know only 5% of the other nodes ids and interests, in order to achieve the required goals. It also assumes there is a heartbeat detection mechanism, which is also used to update a node's neighbors with its degree and target degree.

SpiderCast maintains a separate overlay for each topic. It guarantees that for each topic every node has enough random links to other nodes interested in the same topic. To this end, the neighbor selection is done using two different heuristics, greedy and random coverage. Those heuristics induce two separate overlays in parallel and are merged in the end. Both add one link at the time and keep a set of topics that are not yet K-covered. This is, keep a set of nodes that are not covered by at least K of its neighbors. At each step, the greedy heuristic selects a neighbor who covers the max number of topics from the set. The random heuristic selects a node who would cover at least a topic from the set.

2.4.1 Discussion

	Pre-Constructed Group	Disseminaion	Load Balance
Meghdoot	No	Tree-Based	No
Terpstra	No	Tree-Based	Incomplete
TERA	Yes	Flooding	No
Scribe	Yes	Tree-Based	No
Kyra	Yes	Tree-Based	No
SpiderCast	Yes	Flooding	No

Table 2.2: Publish/Subscribe information for certain metrics. Incomplete load balance represents that try to diminish the burden of being a root node.

We have seen that Publish/Subscribe protocols deal with the routing problem in very different ways. Meghdoot, for example, uses an n-dimensional DHT, from CAN, to organize its routing [22]. It also uses the same overlay for the membership and routing messages. Terpstra 2003 builds the membership substrate using Chord, and each publisher forms a spanning tree that is used for routing its events [43]. TERA [43] uses Cyclon to build a lower layer to organize its peers. It builds multiple layers where nodes subscribed to a topic, those are also grouped using Cyclon. Scribe [11] builds both layers using Pastry. In a sense, it builds a Pastry route for every topic on the system. Kyra [10] is a service network organization for Pub/Sub systems. It creates several groups based on a proximity metric and an upper layer that connects subscriptions in different groups using trees. SpiderCast [13] does not specify a membership structure, but it states that nodes know little about the rest of the system. It uses a separate routing overlay, based on partial views, for each topic.

Table 2.2 shows important metrics for the studied systems. The common practice is to maintain a separate routing overlay for each topic, delivering messages to those pre-constructed groups. In environments with a very large number of subscriptions per node, pre-constructed groups can be hard to manage. Nodes need to constantly exchange messages to update their topic groups. If those types of messages are not organized or efficiently distributed, it can represent a large overhead for the system. Due to the characteristics of our environment churn is not a concern, but constant interest (subscription updates) changes are.

Most of the pre-constructed groups are tree-based approaches. This allows for a better control while delivering messages. At the same time avoids double deliveries, without the need to store message metadata.

It was observed that the systems do not have a viable load balance strategy. Apart from Terpstra, root nodes or central nodes on the multicast groups will be forced to perform more work due to their position. Terpstra performs a better distribution as dissemination trees are built on the fly for each event. Although, if a publisher sends a large number of events for the same topic, child nodes on those trees will constantly receive events without producing any work.

We concluded before that is important to maximize the routing table. Therefore, we aimed for a membership layer, in our system, comparable to the One-Hop. We can discard systems that use unstructured overlays due to their routing tables size. Meghdoot and Terpstra 2003 are the two only systems that use structured membership overlays. Meghdoot uses CAN to build the membership substrate, which uses a routing table too small for our requirements. It also performs its routing in a linear way that prevents back-propagation, resulting in message flow patterns.

In Terpstra 2003 the membership layer is built using Chord, which is also too small for our requirements. Although, its routing mechanism takes a similar approach to our plan. Nodes that subscribe to the same topic are not kept in a dissemination group. Instead, when publishing an event, messages contain a range field of the remaining nodes the message is yet to be delivered to. When a node receives the message, it divides it into several sub-ranges. Each sub-range is sent to the first node that matches that filter inside the sub-range. This way the protocol avoids the overhead of keeping a different message dissemination group for each topic.

Furthermore, by keeping a different dissemination tree for each topic, leaf nodes never perform any work. Plus it introduces a message flow pattern in the system. Randomizing the tree not only forces leaf nodes to perform work, but it also breaks the flow pattern. Bigger routing tables provide more options when randomly selecting the children that will route the events.

2.5 Summary

In this chapter, we analyzed two probabilistic data structures that can help reduce the space overhead in our system. The choice for the best one will be depended on the rate of false positives that we can tolerate. We concluded that the hashing digest used by this structures can obfuscate information. Therefore, we can create filters with a certain level of privacy.

We also studied alternatives for a repair mechanism, such as error correction codes. These are useful at dealing with uncorrelated losses, at different groups of receivers.

We analyzed several peer-to-peer overlay networks. The conclusion we came to is that unstructured overlays are better suited to deal with constant churn and massive node failure, which are not the main concern in the system and scenarios we have envisioned. Structured overlays produce better routing costs at the expense of larger and more rigid routing tables. One-Hop and Two-hop provide the complete tables that inspired our solution. The cost of keeping big tables is associated with churn. Since churn is not predominant in our environment, the costs to keep these tables are reasonable.

Finally, we studied several publish/subscribe systems that used a diverse range of ways to route events in their systems. Keeping pre-assembled groups for each topic is an unnecessary overhead that we believe can be avoided by creating dissemination tree on the go. Moreover, not only do random dissemination trees improve load balance, as they

also increase security properties by avoiding patterns. All components studied helped us to cement our understanding on the problem at hand, and commit to envisioned solution.

Chapter 3

Proposed Solution

Our goal has been to develop a Pub/Sub solution for decentralized environments. Decentralized environments prevent single points of failure and also work as a means to distribute the work between participants.

We propose a solution for two different types of scenarios. In the first scenario, we expect a large number of participant nodes, with decent bandwidth capacities. Participants are expected to stay in the system for limited periods of time, resulting in a moderate amount of churn. Though to their limited sessions, nodes are not expected to perform many changes in their lists of interests, so filter updates will be uncommon. Furthermore, we expected nodes to subscribe to a number of topics that can be stored in small filters

In the second scenario, we propose that a group of participant nodes acts as brokers for a large number of client nodes - the actual event subscribers. The broker nodes are robust, with good connectivity and bandwidth capacities, hence, they do not leave the system. This will cause the number of participants to be lower than the previously described scenario. The client nodes are volatile with short sessions, like mobile devices.

Broker nodes maintain the Pub/Sub based on the interests of the client nodes. Client nodes can receive and publish events, but will not participate in the system, i.e., they do not exchange messages to maintain the overlay, nor do they forward routing events. Filters used by the broker nodes need to be considerably larger, in order to store the wide range of interests gathered from the client nodes and to do so within an acceptable level of FPP. However, since client nodes are volatile, the participant nodes lists of interests are prompt to change frequently, as new clients come and go with different interests. This causes a scenario akin to node churn, that can denominate as filter churn.

It is important to highlight that in neither of the proposed scenarios the participant nodes are mobile devices.

Our system aims to filter out information, distributing the work done by nodes in a

fair manner. It should only deliver messages to a node if it is interested in them. A few exceptions to this rule are acceptable, as we hinted before. We also want to add privacy preserving features to our system. To that end, Bloom filters will be used to obfuscate subscription topics. This allows nodes to receive events they are interested in, without explicitly exposing their subscription list. Nodes that subscribe to very few or specific topics might draw attention to themselves. To mitigate the problem, affected nodes might subscribe to spurious topics. This means false positives play an important role in the work distribution ratio. Furthermore, the event dissemination process is randomized to avoid exposing patterns in the flow of events across the network. To secure the event payload, we will resort to cryptographic functions. These are the key ingredients in which we based the system model.

To accomplish the described proprieties, our proposed solution is structured in two separate layers - a membership layer and on top of that a routing layer. The routing layer builds upon the knowledge provided by the membership layer, whose job is to allow each participant node to know about all the other nodes in the system and their current subscriptions. The membership layer is akin to a one-hop structured overlay, as described in [24], but using our own algorithms, as explained next. The aim of the routing layer is to ensure that each event is delivered to every node that has subscribed it. To that end, events are tagged with topics. Nodes supply a list of topics they are interested in, in the form of Bloom filters, when they join the system. The goal is to achieve event dissemination in a decentralized and fair way that covers all the intended event recipients and preferably no more. Nodes with wide interests, ie., those that subscribe to many topics, are expected to do more work than those that have narrower interests. The meaning of work in this context comprises the matching of events to subscribers and forwarding them to those nodes.

3.1 Membership Substrate

The membership substrate comprises two mechanisms. The job of both is to allow all participant nodes in the system to maintain an up-to-date view of the current state of the overlay network. The view contains the membership of the system, i.e., it contains all participant nodes and their respective filters. The main mechanism consists in having selected nodes broadcast new node arrivals. This happens periodically and resorts to random dissemination trees built on-the-fly. A second low priority mechanism is used to recover messages lost during the broadcast phase due to node failures. This mechanism relies on periodic epidemic exchanges between pairs of nodes selected at random.

To structure the overlay, nodes are organized in a key arrangement as a ring modulo of 2^{128}bits. The ring is equally partitioned into slices, where the node with the smallest key in each will act as its leader. The leader is responsible for collecting messages over a period of time after which it initiates a broadcast. The messages received during that period, are aggregated into a single message containing all membership changes for the

slice. The collection avoids network spikes or congestions by frequent changes to the membership. It also becomes more feasible to establish a partial ordering on larger and less frequent messages.

The first step for a new node its to generate its unique key by hashing its IP address. Before participating in the system, the new node needs to acquire the full view of the system. This can be achieved by messaging a previously acquired contact node. The contact node is responsible for replaying with the system list of endpoints. The filter download is separated from the endpoint download to avoid creating considerably large messages. If the new node fails to obtain the endpoint information from the contact node, it can resend the request or acquire a new contact node. After obtaining the system endpoints, i.e., the view of the system, the new node can select any random node and query for the filters. If the system has a substantial size, the filter download can be divided among several nodes, requesting different parts to each node. Any part that fails to be downloaded can be requested again to a different node.

Using the newly acquired view of the system, the new node determines whom the leader of its slice is, sending it a request to join the membership. Nodes that want to update their filter, also contact their slice leader. It is important to notice that the leader of the slice is locally recalculated by the node, each time it needs to contact it. When the system is initiating and it is too small, there might be no given leader for a specific slice. In that case, the leader of the previous adjacent slice is used, and so on until a leader is found.

The new node only considers itself in the system once it receives a broadcast. That means its leader has emitted a broadcast with the new node information, implying that other nodes will now be aware of the new node. If after some period of time the node does not receive any broadcast, it recalculates the leader and tries to join the membership again. Node failures are not disseminated in the system, to avoid disseminating false information about nodes that might still be alive but were considered dead over a faulty connection. Consequently, if a leader is faulty, nodes trying to join or nodes trying to update their filters will end up recalculating a new leader. We assume that for brief periods of time a slice might contain more than one leader. Those situations do not cause any problems since nodes only calculate a leader at the time, and eventually they converge to a single leader.

Connections between nodes are made over TCP, so the protocol can be used as a fail detector. If the connection to a determined node fails over a certain number of times, the node can be considered as failed. This means that while propagating a broadcast, the failed node will be skipped. It also means that when sending the system view to a new node, nodes in the endpoint list are tagged with a failed flag. If any broadcast contains an update from the failed node, then it can be considered alive once more. Otherwise, nodes that are considered failed over a long period of time can be discarded from the system view.

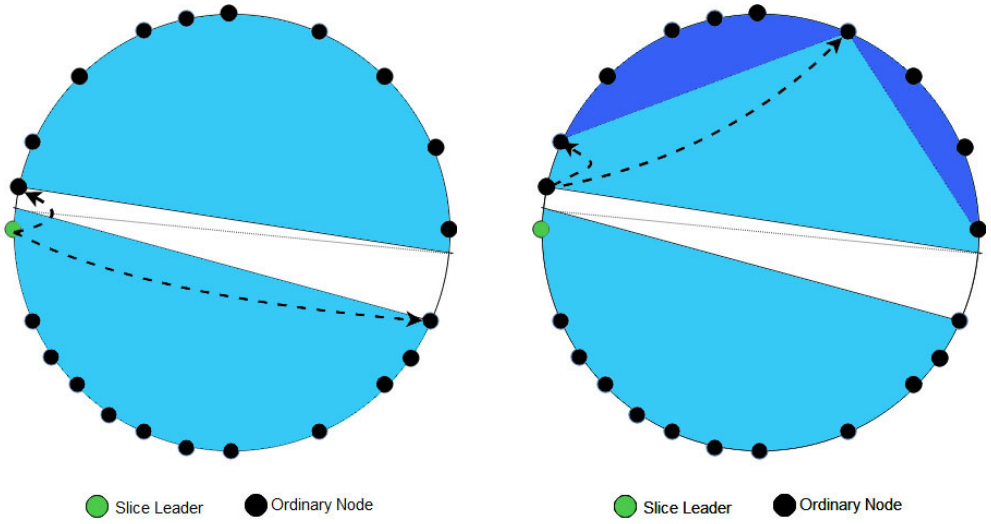


Figure 3.1: Membership substrate broadcast dissemination with fanout 2 for one of the tree branches. Highlighted zones represent the remaining key space for each sub range. For simplicity not all dissemination paths are shown.

Algorithm 1 *Membership Broadcast*

```
1:  Procedure newBroadcast do:
2:      if accumulatedUpdates ≠ ⊥ then
3:          updates[] ← accumulatedUpdates;
4:          accumulatedUpdates ← ⊥;
5:          timestamp ← {(key, counter)};
6:          counter ← counter + 1;
7:          range ← getFullRangeView();
8:          range ← randomRotation(range);
9:          msgID ← uniqueMsgID();
10:         trigger Broadcast(msgID, range, updates[], timestamp);
11: Upon Receive(Broadcast, msgID, range, updates, timestamp) do:
12:     trigger UpdateView(updates);
13:     receivedBroadcasts ← receivedBroadcasts ∪ {(msgID, updates)};
14:     clock ← clock ∪ timestamp;
15:     ranges[] ← divideRange(range);
16:     foreach r ∈ ranges[] do
17:         node ← findCandidate(r);
18:         if node ≠ ⊥ then
19:             newRange ← [node.key, r.max];
20:             trigger Send(Broadcast, node, msgID, newRange, updates, timestamp);
```

3.1.1 Broadcast

In each slice, the node with the smallest key will be considered its leader. Leaders are responsible for collecting membership updates from that slice, for a certain period of time, after which they initiated a broadcast. They are also responsible for partly ordering the collected updates by using version vector timestamps. Broadcast periods are independent

between leaders and therefore not synchronized.

The broadcast starts by taking the full node key range and rotating it by a random offset. This step guarantees that the broadcast tree is different each time, distributing the work in a fair manner between all nodes. Moreover, by changing the dissemination tree often, the position of each node in the tree will tend to alternate between *interior node* that does work and *terminal node* that does not. The initial range is then divided into sub-ranges. The division process is done by the remaining amount of nodes in a key range, and not by the dimension of the range. Since it is not certain that keys are equally distributed across the key range, dividing it by the range dimension results in uneven work, as some ranges will have more nodes in them compared to others. The number of divisions (fanout) can be established by a system parameter or calculated dynamically. A dynamical fanout means that depending on the amount of work performed by the node, the fanout can be increased or lowered, helping nodes to achieve a better balanced work ratio.

For each sub-range, a candidate node is selected. The candidate is the first alive node of that sub-range. The remaining key range is updated by stripping the keys leading up to the candidate node's key. This step reduces the scope of the distributed search as the dissemination of the tree broadcast proceeds along the tree. Finally, the leader sends to each candidate the list of membership updates it has collected, the updated key range and the broadcast timestamp. Figure 3.1 shows an example of the division process. The pseudocode for the membership layer can be found in Algorithm 1. Nodes that receive the broadcast must repeat the division and candidate selection process, forwarding the broadcast until the key range is exhausted.

Once the new node receives its first broadcast, it can consider itself in the membership, allowing it to accept new node arrivals, updates, start its repair mechanism and begin to emit its own routing events. The updates, contained on the broadcasts, are added to the receiving nodes local views and stored in a set of received broadcasts.

3.1.2 Membership Repair

The membership layer was designed to provide best-effort delivery, i.e., it does not provide any guarantee that messages are delivered. Therefore, a repair mechanism was deemed necessary.

As each broadcast is tagged with a unique timestamp, nodes build a version vector with the received broadcasts, which allows to establish a partial order of the broadcasts. To ensure an up-to-date view, nodes periodically select a random partner and exchange their version vector. By comparing version vectors, nodes can detect missed broadcast messages between them. Since nodes are selected at random for this step, lost messages are propagated epidemically. Therefore, it is safe to assume nodes are up-to-date with the view of the entire system, which allows them to safely perform the candidate selection step, without skipping nodes.

3.2 Routing Layer

The routing layer algorithm is quite similar to the one described for the membership substrate. The key difference is that, instead of issuing broadcasts that target all nodes in the system, each published event needs to be delivered to a fraction of the nodes, ie., those interested in a particular topic of each particular event. Therefore, when the dissemination tree is built on-the-fly for each event, at each step, intermediate nodes are selected among the remaining ones only if they subscribe to the topics of that particular event. This requires leveraging the information provided by the membership substrate, which ensures every node knows about all other nodes in the system and the topics they subscribe.

Figure 3.2 demonstrates how the mechanism operates. The publisher node selects the entire key range and rotates it by a random offset. The initial key range is divided into sub-ranges, according to the desired tree fanout, irrespective of the membership fanout. In the example provided in Figure 3.2a, a fanout of 2 is used. For each sub-range, the publisher searches for a candidate node. The candidate is the node with the smallest key, whose subscription matches the event topics [1]. Each candidate node is sent a message containing: the remaining updated key range, starting just past its own key, the event payload and list of topics. This recursive division and the event matching process is repeated until all keys are exhausted.

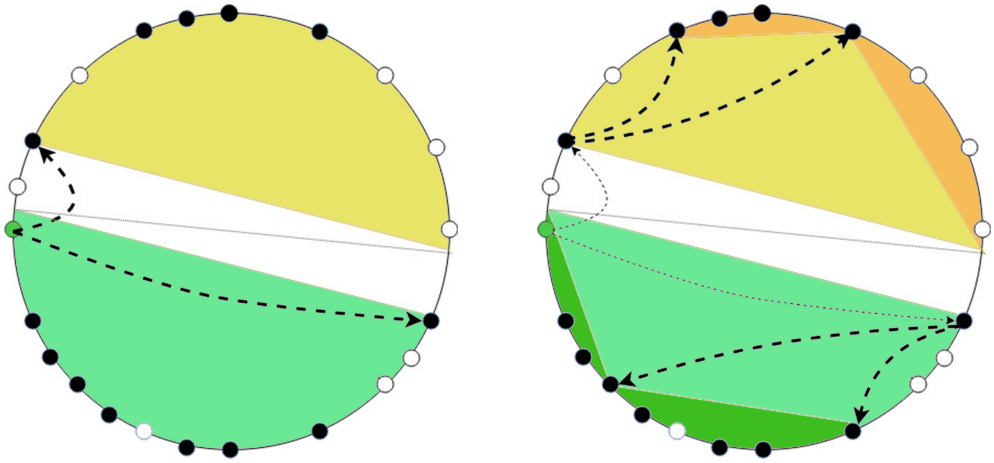
The initial rotation not only provides the desired load balance proprieties described before, but it also mitigates any patterns that might be present in the flow of events. The resulting random dissemination tree ensures that successive events reach each subscriber following many different paths. This improves the fairness of the process.

A new node might be a target in the routing layer forwarding process, as soon as its membership leader propagates a broadcast containing that new node. Hence, the node can receive a routing event while it is still downloading the system filters. In that case, if the node has to evaluate a node whose filter it still has not received, it will consider the missing filter as a match and forward the event to that node, which might result in a false positive for the receiving node.

3.2.1 Failures on the Routing Layer

Similarly to the membership layer, the routing layer is also best-effort, so the use of a proper repair mechanism is mandatory. Without it, if a node fails to propagate an event, the remaining of its sub-range will never receive that event. Depending on the fanout, it could mean that up to half the system does not receive that event. Although, event messages do not travel through all nodes, but only between the nodes that subscribe to the event topics. Furthermore, the overall number of event messages sent is, which are

[1] Since events can carry more than one topic, event matching can have multiple acceptance levels. The event can be matched with only one common topic, or if all topics match the filter or if a percentage of the topics match the filter.



(a) Dissemination step 1 (b) Dissemination step 2

Figure 3.2: Example of event dissemination tree with a fanout of 2, initiated at the green node, showing the first two levels. Shaded areas exemplify the amount of work remaining that is passed from parent to child node. White nodes do not match the event filter.

Algorithm 2 *Routing Broadcast*

```
1:  Procedure New Routing Event(topics,payload) do:
2:      shards[] ← encode(payload);
3:      shardID ← 0;
4:      msgID ← uniqueMsgID();
5:      foreach shard ∈ shards[] do
6:          range ← getFullRangeView();
7:          range ← randomRotation(range);
8:          trigger RoutingBroadcast(range, topics, msgID, shardID, shard);
9:          shardID ← shardID + 1;
10: Upon Receive(RoutingBroadcast, range, topics, msgID, shardID, shard) do:
11:     trigger receiveShard(msgID, shardID, shard);
12:     ranges[] ← divideRange(range);
13:     foreach r ∈ ranges[] do
14:         node ← findRoutingCandidate(r, topics);
15:         if node ≠ ⊥ then
16:             newRange ← [node.key, r.max];
17:             trigger Send(RoutingBroadcast, node, newRange, topics, msgID, shardID,
    shard);
18: Procedure receiveShard(msgID, shardID, shard) do:
19:     if msgID ∉ decodedEvents ∧ (shardID,_) ∉ receivedEvents[msgID] then
20:         receivedEvents[msgID] ← receivedEvents[msgID] ∪ {(shardID,shard)};
21:         if #receivedEvents[msgID] ≥ DECODE then    ▷ necesasry number of shards.
22:             trigger decode(receivedEvents[msgID]);
23:             decodedEvents ← msgID;
24:             receivedEvents[msgID] ← ⊥;
```

not aggregated, would make it difficult. Hence, dealing with failures in the routing layer

cannot be exactly achieved in the same way as in the membership layer.

To prevent event loss we leveraged the erasure code Reed-Solomon, meaning the event message is sent several times, i.e., with some level of redundancy, depending on the used number of shards. The probability to recover a message will be associated with the total number of shards and the number of data shards. Furthermore, the use of this method has advantages when compared to simply duplicating the message for redundancy. By dividing the message into small shards, the system can keep a considerable redundancy level, without an extreme increase in message size. The same would not be true in a redundancy model made by simply multiplying the event messages.

For the method to be effective, the message shards have to be sent by different paths each time, i.e., different dissemination trees. If it were to follow the same path for each shard, then it could fail at the exact point each time. Each receiving node needs to store the received shards until it has enough shards to decode the event payload, after which the node can discard the shards. To avoid dealing with the shards that have not yet been received, nodes keep a list of already decoded events. When the node receives the shard of an already decoded event, it still proceeds to forward that shard, as usual. But it avoids storing the shard by checking the list of decoded events. The pseudocode for the Routing layer with Reed-Solomon can be found in Algorithm 2.

When a node fails to propagate a message, it means a portion of the dissemination tree does not receive the shard. Nodes will fail to decode an event if they are contained between a number of failed trees bigger then the required number of shards necessary to decode the event. In a system running for enough time, this can happen since trees are randomly rotated but not guaranteed to be disjoint.

Furthermore, the erasure code can only be applied to the payload of the event message and not the full message. Making the entire event message redundant through this method would mean the event propagation would not continue until a node receives the minimum pieces. The reason for that is that the remaining range and the event topics would be impossible to read until the message was reconstructed. Waiting for the rest of the shards would also be impossible since the message needs to be forwarded in different paths, and thus the different shards would never reach the remaining nodes.

The use of Reed-Solomon as a failure prevention mechanism as the benefit of not adding additional latency. Although, it comes with a large cost in the number of routed messages. Alternatively, to use a repair mechanism based on lost events, like the mechanism used in the membership layer, nodes would have to find partners with similar interests. Partners could be found by leveraging the information provided by the membership layer and performing bitwise AND operations between filters. Events would also need to be enumerated and stored for a period of time, that would guarantee that the correct comparison between missing events could be made. Furthermore, the use of one mechanism does not strictly exclude the other. Both mechanisms could even be used complementary, using the latter as a backup mechanism.

3.3 Filters

Routing events are tagged with an obfuscated version of topics. Filter comparison is done by digesting the obfuscated topics through the Bloom filter set of hash functions, and checking if the returned bits are marked as 1's in the filter. A downside is that, as the event carries more topics, the bigger the messages will be. Alternatively, instead of a list of obfuscated topics, the event could carry a Bloom filter with the topics already inserted in it. The filter matching operation could then be performed by comparing filters using a bitwise AND operation. Although, as the number of topics in the event increases, so does the number of ON bits on the filter. This would make it difficult to perform a filter match based on a certain number of present topics, hampering the ability to make successful comparisons between two filters and increasing the number of false positives.

Tagging the event with the filter, instead of the list of obfuscated topics, would force participant nodes to use strictly identical filters. Otherwise, the filter comparison would not be possible. By imposing a filter size in the entire system, the appropriate filter size must be chosen carefully, in order to accommodate participant nodes with wide interests. The use of large filters impacts the cost performance of the membership layer, making it more expensive. Although, if the filter is too small, it will cause false positives in the routing layer. False positives represent unbalanced work to the system, as nodes receive events that do not match their filters and have to perform work to route those events.

If the event is tagged with the list of obfuscated events, the nodes might use filters with different sizes and parameters. Though to the information provided by the membership layer, each node has the necessary information to calculate the correct bit positions for different filters. Hence, nodes can use filters adjusted to the size of their interest range. Nodes with narrow interests can keep small filters, lowering the cost to maintain the membership layer.

In most cases, is beneficial to reduce the number of false positives, as much as possible. However, nodes that subscribe to very few or very specific topics may draw attention to themselves because of that. Affected nodes can mitigate the problem if they are willing to accept some false positives by subscribing to spurious topics. This would help obfuscate the node's specific interests, at the expense of increasing the amount of work done.

3.3.1 Filter Privacy

Bloom filters resort to hash functions to calculate the position of the bits in the array. A common characteristic in hash functions is that they are irreversible, so in a certain way they can obfuscate topics in the system. Although, relying only on the Bloom filter built in hash functions is not enough to ensure privacy. Before each topic is inserted in the Bloom filter it must be further obfuscated by a keyed-hash message authentication code, HMAC. The HMAC process involves the use of a cryptographic hash function and a secret cryptographic key. This feature can even allow the system to support public and

private topics at the same time. The obfuscated version of topics, present in the event, is also obfuscated by the HMAC function.

A feature of Bloom Filters is that false positives do not increase with the number of available topics in the system, but with the filled filter percentage (% of ON bits). Therefore, private topics will not increase the level of FPP on the system. However, we are assuming that events with private topics carry payloads ciphered with different keys, meaning only the desired nodes will be able to decode those payloads.

The usefulness of the Bloom filter is limited to compacting and securing the representation of the subscriptions. The payload will need to be secured as well. Standard cryptography functions can be used to encrypt the payload. We assume nodes will agree on the keys using some external procedure to our system.

Chapter 4

Implementation

Following the model described in the previous chapter, this chapter is dedicated to making a thorough explanation of how the implementation of each component was achieved. The proposed model was implemented as both a real-world application and a simulation environment. Even though we implemented a working real-world application, its design was focused on obtaining performance metrics with the intent of calibrating our simulation environment, to provide a meaningful performance evaluation of our solution in large-scale scenarios. Nevertheless, the components designed for evaluation and telemetry can be disabled or easily adapted, allowing for a more regular utilization.

This chapter is structured as follows. A first section is dedicated to explaining some overall details of the implementation. It is followed by a description of the membership and routing layer implementation. We also present the explanation of some components that were implemented in order to access the system metrics, some possible system optimizations, and the simulated environment.

4.1 Overview

The system was implemented in Java. This decision was made due to the language ease of use and the wide range of libraries available.

To establish the communication between nodes we used a toolkit named Akka [48] [1]. Akka simplifies the construction of concurrent and distributed applications on the JVM. It also has emphasizes on an actor-based concurrency model based on asynchronous messages. In computer science, actor models are known for making local decisions based on received messages. Actors can modify their state, but can only affect each other through messages. As so, the actor model avoids the need for locks. The model is ideal for our membership and routing layers since those must constantly exchange messages without

blocking themselves. Each layer is associated with its own actor since both layers only have to establish communication with the same layer on other nodes.

The routing layer does not perform writes on the node's database, only read operations. Therefore, and for simplicity and efficiency purposes, the database is shared between the actors in a node, using thread-safe data structures.

By default, Akka uses TCP/IP as the communication protocol. Using Akka/TCP allowed us to use the latter as a failure detection mechanism. All messages exchanged between different nodes extended a Message class of our own making. The class contains a message identifier, unique for each message the node has sent, and the node's key.

Akka accommodates message passing between actors in the same JVM as a direct reference passing [2]. Although, messages that leave the JVM (to reach an actor running on a different host) need to undergo serialization. By default, serialization is done by Protocol Buffers for internal Akka messages and with Java serialization for the remaining type of messages, i.e., messages that leave the JVM.

Unfortunately, Java serialization is known for its bad performance, efficiency and for not being secure [32] [27]. To that end, we employed an external serialization library denominated Kryo. Kryo delivers high speed and low size binary object graph serialization framework [39]. However, applying Kryo directly to the Akka actors was not a trivial task and messages leaving the JVM still end up being serialized with Java. To face that problem, every message leaving the JVM is serialized with Kyro to a byte array. When leaving the JVM, Akka serializes the byte array with the default Java serializer. This solution is not ideal. Although, it is more space and time efficient than singly using Java serialization. In this case, the Java serialization is only used as a wrapper to the already serialized byte array.

Objects serialized with Kryo must be registered with a unique numerical identifier before they are serialized. Despite not being strictly necessary to use Kyro in that manner, it is required for our system. Otherwise, Kyro will attribute free identifiers to objects as they are serialized for the first time, leading to inconsistent identifiers between different nodes.

Furthermore, Kryo serializers are not thread safe and configuring a Kyro instance is relatively expensive. For those cases, Kryo provides a thread pool from where instances can be safely concurrently pooled when needed. The max number of instances stored is a parameter of the pool. When more instances of Kyro are needed than those that can be stored, they can be initialized but will later be discarded.

To allow the system to be deployed and tested in diverse environments, all system parameters can be changed through a text configuration file. On startup, nodes will read the configuration file and override default parameters. For the correct behavior of the system, it is important that all nodes run with the same configuration.

4.1.1 Filter

For the probabilistic data set, we chose the Bloom Filter over the Cuckoo Filter. The reason for our choice can be found in the Annex I. We chose an open source Java Bloom Filter implementation available on GitHub [38].

The chosen implementation allowed the creation of Bloom Filters given: the size of the bit set and the expected number of elements; or the false positive probability and the expected number of elements; or an explicit declaration of the number of bits per elements, expected number of elements and number of hash functions to use. To further accommodate the needs of our system, we extended the Bloom filter to be initiated with the filter size in bits and the expected false positive probability (FPP) as parameters. This was possible due to the direct use of the Bloom filters theoretical formulas.

It is important to notice numerical values given by the Bloom Filter theoretical formulas, such as the number of bits per element and the expected number of elements, have to be rounded to integer numbers. This detail creates a small difference between the desired properties of the filter and the final ones. It mostly translates to the filter having an FPP below the desired one.

Provided a certain size in bits, Bloom filters allow for some flexibility in the number of topics that can be stored. To give a sense on the capacity of Bloom Filter, filters were created with different bit sizes and different sizes on the set of hash functions. Topics were inserted into to the filters until a certain level of FPP had been reached, as shown in Table 4.1. The difference in the number of inserted topics is small for 1% FPP, although it is considerable for 5%. This happens since Bloom filters seem to peak their performance when they are close to being 50% filled, as shown by Table 4.2. The table shows the % of ON bits when the filter reached the desired FPP.

The Bloom filter implementation stores the filter as a bit set using the java.util BitSet library. Kryo has a known problem serializing that BitSet implementation, so the Bloom Filter was further extended to use the Apache OpenBitSet implementation [4].

Events contain a list of obfuscated topics, which are used to calculate if a topic is contained in the Bloom filter. The number of topics the filter has to contain for it to accept an event is a system parameter. We designed our implementation to use filters with the same parameters. So in our implementation, it is mandatory for nodes to use the same filters for its its correct behavior. As explained before, the Bloom Filter calculates the k bit positions and check if all are set to 1.

Each node stores an internal list of the topics it subscribes too. When performing an update to its filter, the node updates the corresponding bit positions of the new topic and sends the new version of the filter to its leader. It also stores the new topic on the list of subscriptions. Bloom filters do not allow topics to be removed from the bit set. Nodes have to deal with the additional false positives caused by unwanted topic subscriptions or rebuild the filter. Using the internal list of topics, the node can calculate a threshold FPP and rebuild the filter when it surpasses that value.

	256 bits		512 bits		1024 bits		16Kbits		32Kbits		64Kbits	
# Hash	1%	5%	1%	5%	1%	5%	1%	5%	1%	5%	1%	5%
10	26	35	52	70	103	139	1595	2163	3190	4325	6380	8650
7	27	39	54	78	107	155	1668	2412	3336	4824	6672	9647
5	26	41	52	82	104	164	1625	2551	3250	5101	6499	10201

Table 4.1: Number of topics inserted in different sized filters according to different FPP.

	256 bits		512 bits		1024 bits	
# Hash	1%	5%	1%	5%	1%	5%
10	60.5%	72.6%	64.8%	75%	64.2%	74.9%
7	49.6%	61.7%	53.7%	66.2%	51.6%	65%
5	39.8%	53.9%	40.8%	57%	40.7%	55.5%

Table 4.2: Percentage of filled filter width (% ON bits) according to different FPP.

To ensure a certain level of confidentiality, before each topic is inserted in the filter, it is obfuscated through a keyed-hash message authentication code. We use the SHA-512 HMAC function, but any other compatible HMAC could be used. Form the resulting digested bits, only a few are used. The number of selected bits is a system parameter and non selected bits are discarded. Discarding bits reduces the overhead of the obfuscated topics. Although, only selecting a small number of bits can cause false positives, as different topics could be digested to the same selected bits. Depending on the variety of topics used in the system, more bits can be added to avoid those situations. By default, we use 32-bits, which is enough to ensure there are no topic collisions on the tested environments. Finally, the resulting used bits are digested through the Bloom filter set of hashes and computed to their corresponding positions on the bit array.

4.2 Membership Layer

4.2.1 Joining the Membership

The first step for a new node joining system is to calculate its unique key. The key is obtained by digesting the node IP address through an MD5 hash function, although any other hash function could be used. The result is then transformed into a numerical representation, stored in a long value. We deemed the long value to be sufficient to store the unique key, without colliding with other nodes keys. In a real deployment, longer keys might be considered.

We assume the new node has access to a contact node inside the system. It uses the contact node to request the list of endpoints in the system, i.e., the system's view. The contact node replies with the list of endpoints and with a version vector that tracks the membership broadcasts seen by the node. Each endpoint contains: the node's unique key;

its IP address; last time it was seen alive; a boolean representation if it is considered alive or not; and the number of times the connection to that node has failed.

The endpoints are stored by the new node on its local database. The node also sets its local version vector to the same version sent by the contact node. This prevents new nodes from performing repair operations on broadcasts older than them and it allows the system to move forward and discard older unnecessary information.

The message to join the membership contains the node's key, its IP address, and the filter. Since in our implementation all participants have to use filters with the same parameters, there is no need to send the entire filter structure. Only the bit set needs to be shared.

The system operates in an asynchronous manner. Hence, new nodes do not receive any direct message from the leader confirming that they have joined. This means that a node only considers itself in the membership, once it receives a broadcast message. Receiving a broadcast means that its leader has broadcasted a message that contains the new node, and other nodes have added the new information to their views. In case the new node does not receive any broadcast for a certain period of time, it can resend the join request to its leader. As the connection is made over TCP, any connection failure can be registered by the node. If the connection to a node fails over a parameterized number of times, then the node can consider that other node as failed. Failed node are skipped when calculating the leader or searching for the next node to continue the dissemination process. Therefore, nodes will end up recalculating a need leader.

4.2.2 Filters Download

Filter download begins after receiving the endpoint list. The filters can be downloaded in their whole or dynamical divided into parts according to the system size. These parts have no correlation to the slices in which the system is divided. For each part the node sends a **Filter Download Request** message to a random node in the system. These messages contain the range of keys from which the new node requested the filters. Nodes answer to the **Filter Download Request** message by sending the filter of each node contained in the range.

Each received filter is matched in the new node's view to its corresponding endpoint. Until all the filters are downloaded the **Filter Download Request** message can be periodically resent to random nodes in the system, requesting the missing filters. At this point, the node might already have joined the membership. This means that even though the new node might still not have all the filters, it can be chosen to propagate a routing layer event. In that case, any missing filter will be evaluated as true, and the event forwarded to that node, which might cause false positives. Other possible solutions are discussed in Section 4.5.

Filter download messages do not need to be tagged with the version vector. If the new node downloads a filter that in the meantime has been updated via a broadcast, it can

ignore the downloaded filter. If the downloaded filter is the same or an older filter, then the new node already has the last version through the broadcast. If the downloaded filter is a new version of the filter, then the new node will eventually receive that version via a broadcast or a repair message.

4.2.3 Broadcast

Nodes that wish to update their filter send a **Update Filter** message to their corresponding slice leader. Slice leaders aggregate update messages and join requests for a parameterized period of time before disseminating them. By aggregating several messages into a larger message, we avoid the waste of bandwidth that would be caused by the overhead of the communication protocol header (Akka/TCP/IP). It also ensures that the membership mechanism robust enough for the correct operation of the routing layer.

The key range works as a circular ring of increasing key order, were the maximum key leads to the lowest key. The key rotation is performed by using a random uniform distribution that selects a key as the starting point of the ring. The key range is then divided into equally sized sub-ranges, according to a parameter fanout. The sub-range division is performed with the remaining amount of nodes in the key range. For each sub-range, the leader needs to calculate whom the candidate node is. As the key ring is ordered by increasing key value, the search can be done by iterating the nodes until one matches the criteria. To propagate the membership broadcast, the candidate only needs to be an alive node. The remaining node key range of that sub-range is updated by stripping the keys leading up to the candidate node's key. To each selected candidate a tree broadcast message is sent, containing: the updated correspondent key range, the list of updates the leader has aggregated, a version vector timestamp.

When a candidate node receives the broadcast message, it repeats the division process until the key range is exhausted. If it is the first time a node has received a broadcast, it updates its state to **joined**, allowing it to accept new node arrivals, updates, start its repair mechanism and begin to emit routing events.

After receiving a broadcast message and before propagating it through the rest of the network, the node processes the list of updates contained in the broadcast. Leaders receive their own broadcast messages as any other common node, i.e., they do not update their view before broadcasting. The version vector timestamp present in the broadcast is added to the local clock of the node. Since the system operates in an eventual consistency model, simultaneous updates can happen. Although, those should not be considered a problem since nodes will only send updates to a slice leader at a time. Even when for brief periods of time there is more than one leader per slice, nodes are only sending updates to the one leader they consider correct, hence, leaders broadcast different updates. If a node that was previously considered failed is present in the update list, it is then considered alive. The number of times it has failed is reset, and its last seen alive timestamp refreshed.

4.2.4 Repair Mechanism

Failures can cause inconsistencies in the nodes membership view. To face those failures the membership layer runs an anti-entropy mechanism in the background to recover missed messages. Nodes perform pair-wise update exchange rounds from time to time to try to detect missed messages. Exchange nodes are chosen at random. This means missed broadcasts will be propagated epidemically throughout the system.

When the repair process starts, the node sends a **Repair Request** message to a randomly chosen node. For simplicity purposes, let us call the node that starts the repair node A and the randomly selected node B. The repair request contains the version clock of node A. When node B receives the message it compares its local clock to the one send by A. If both clocks are equal, then both nodes have the same view and the repair process ends. Otherwise, if B detects that A's clock has timestamps that he does not, it will ask A for those broadcasts. Node B also checks is A is missing any timestamp. If it does, the replay message will contain both the list of B missing timestamps and a list of A missing broadcasts. Node A will then replay to node B with B's missing broadcasts. Missed broadcasts are then inserted into each node's local database.

4.2.5 Connection Failures

Akka allows actors to be directly notified if an open connection to another actor fails. In case there is no open connection, a failure to open a TCP channel will also allow us to detect failures. By default, an Akka actor will not receive this type of information. To do so actors need to subscribe to a special type of event denominated *Disassociated Event*. From that special event, it is possible to retrieve the IP address of the connection that failed. Only one actor needs to subscribe to this type of event. Any *Disassociated Event* generated by other actors will be received by the subscribing actor, which is the membership layer actor.

Nodes keep in their database a counter for each time a node has failed. This is, each time they receive a *Disassociated Event* for that node address. If a node fails over a parameterized number of times, it will then be considered dead. The node will only be considered dead in local views. Node failures are not propagated as updates to the membership.

4.3 Routing Layer

The routing algorithm is similar to the one described in the membership layer. Although, routing events carry an event payload and a list of obfuscated topics. Since the system was designed to obtain valid experimental results, the payload is simulated as a randomly filled array of bits, with a parameterized size. The array is randomly filled to avoid possible compressions the serialization process might perform on an empty array. Because we are considering that participant nodes can act as servers to the actual subscriber nodes, topic selection is performed at random from any of the topics being used in the

system. Meaning nodes can start the propagation of an event containing topics they do not subscribe to. Chosen topics are obfuscated through the HMAC function before being inserted in the event.

To emit an event the node starts by rotating and dividing the key range, similar to the membership broadcast. The fanout used in the routing layer can be different from the one used in the membership. Akin to the membership algorithm, for each sub-range the node needs to find a candidate node. The candidate node has to be an alive node with a filter that accepts the event topics. The search process can be achieved using the global view built by the membership layer, which contains every participant node's filter. The number of topics a filter needs to accept in order to be considered a candidate is a system parameter. Once found, the remaining key range is updated by stripping the keys leading up to the candidate node's key. The event is then forwarded to each candidate node, containing: the list of obfuscated topics, the payload, and the updated range. Receiving nodes repeat the sub-range division operation, finding the next candidate nodes until the key range is exhausted.

4.3.1 Dealing with Failures

To deal with failures, we adopted the error correcting algorithm Reed-Solomon. In terms of implementation, we selected the Backblaze Java Reed-Solomon library, available on GitHub [6]. The library requires 3 configurable parameters for its operation: the number of data shards, the number of parity shards and the total number of shards. Data shards represent the minimum needed shards to reconstruct the original message. Parity shards are the extra shards. The total number of shards parameter must be equal to the sum of the data shards with the parity shards. The application of this method means that the message will be sent a total number of times corresponding to the total number of shards parameter.

Table 4.3 and 4.4 show the correlation between the number of shards needed for reconstructing the message and their size. More examples can be found on Annex II.2. As the tables show, if the message could be reconstructed with only one of the shards, then each shard had to contain the full size of the initial message. Moreover, if the message needed all shards to be reconstructed, then each shard would contain approximately the size of the initial message divided by the total number of shards. Neither, of these extremes are interesting as needing all the shards would mean that no message could be lost, rendering the processes useless. On the other hand, recovering the message using only one shard would be the same as just duplicating the message. The total transmission cost present in the tables portraits the total cost of broadcasting the shards, taking into consideration the rest of the message contents and including the header size. For a practical example, we considered other message contents and header have a size of 100 bytes.

When a node decides to emit an event, it processes the payload through the Reed-Solomon encoder. This results in the payload being divided into a number of byte arrays corresponding to the total number of shards. For each shard, the key space needs to be randomly rotated and divided into sub-ranges. The event message is sent to the selected candidate nodes, caring the previously explained event contents plus a numerical id of the shard and the shard. The emitter node performs the division and rotation operation for each shard.

Receiving nodes operate as usual, forwarding the events to the next candidate nodes. However, nodes need to keep two different structures to deal with the received shards. The first structure contains a list of received shards for each event payload that still has not been reconstructed. The second structure keeps a list of already reconstructed event id's. The node reconstructs the event payload as soon as it receives the necessary number of shards. The identifier of that event is stored in the list of reconstructed events. Shards used to reconstruct the event are then discarded since they are no longer necessary. The list of events already decoded prevents the node from dealing with shards of events that have already been reconstructed, allowing the node to immediately discard those shards. Although, the node still needs to forward those shards.

Needed Shards	Individual Shard Size	Sum of Shards	Cost with headers
1/3	100	300	600
2/3	50	150	450
3/3	34	102	402

Table 4.3: Reed-Solomon reconstruction for 100 byte message using 3 shards in total.

Needed Shards	Individual Shard Size	Sum of Shards	Cost with headers
1/4	100	400	800
2/4	50	200	600
3/4	34	136	536
4/4	25	100	500

Table 4.4: Reed-Solomon reconstruction for 100 byte message using 4 shards in total.

4.4 Evaluation Components

Since our real-world implementation was designed with experimental evaluation in mind, we implemented a contact server and a telemetry component able to register the node's communication costs. Both these components can be disabled and are not required for the correct behavior of the system.

Every message that leaves or enters a node is passed through a telemetry class. The class is responsible for registering all outbound and inbound communication for that node. It registers the number of sent and received messages, as well as, the amount of

uploaded and downloaded bytes for each type of message, including TCP/IP message headers.

After a node is done executing, it stores its telemetry information to a spreadsheet file. To avoid concurrent accesses to the file, each node keeps its unique file identified by its key. During its execution, a node can write the telemetry to the file several times. Each write represents a different row in the spreadsheet and is completed by a timestamp column. This allows us to take sample executions over that node's telemetry. When the system is done executing, a script compiles all the spreadsheets into a single one and generates system statistics for that execution.

4.4.1 Contact Server/Oracle

To help us start and monitor the system, we designed an external entity called Oracle. The Oracle server keeps a track of all nodes that entered or left the system and is assumed to always be reachable. It provides contact addresses for nodes that desire to join the Pub/Sub. Therefore, a node that desires to enter the system will request a contact node to the Oracle. If there are no nodes in the system, the Oracle answers with an empty contact. Otherwise, it answers sending the address of a random node in the system. In case the contact node fails to respond, the new node can request another contact node from the Oracle.

To avoid storing nodes that might have still not joined the system or nodes that might fail in doing so, the Oracle does not register new nodes straight away. Nodes have to send a message to the Oracle once they enter the system, only then are they registered as contact nodes.

The Oracle also works as a network monitor. To help us conduct the experimental evaluation it can send messages that alter the system behavior. As so the Oracle can: request nodes to start taking a sample of the network; introduce failures in the system; inject publish/subscribe events, and kill nodes in the system. Again, these operations are only to help us test and register the system telemetry and would be disabled in a real-world scenario.

4.5 System Optimization

This section is dedicated to some improvements that could be made in our solution.

When a node receives routing events and it is missing filters, there are other possible solutions that could have been approached.

- Extending our approach, a flag could be added to the event message. On the presence of that flag, the receiving node would send its filter back to the node missing its filter.

- If the node that receives the event has its filter list incomplete, then it could return the event to the sender. The sender would discard the node for this round and forward it to the next candidate. This solution would not cause false positives as the previous one, but it would add an extra delay to the system.
- If the node that receives the event has its filter list incomplete, it could save the event and wait for the filter list to be complete. This solution could potentially be harmful to the system, as events could get stuck for unwanted periods of time.

4.6 Simulator

The goal of our system is to run in deployments representative of edge computing scenarios. Namely, our deployments will target systems with a large number of nodes and systems with a modest number of nodes but with an immense number of events. Given those proprieties, we used simulation to test our system components and metrics on larger systems.

The simulator software had been developed before the start of this work. It had been developed to provide a membership layer with characteristic as the one developed in this work. No major changes were required, although, if a node desired to update its filter, it would have to leave the system and reenter with a new filter. On the real-world implementation, a node can directly contact its leader with an updated filter, so the simulator was adapted to those characteristics.

The routing layer provided a message delivery system based on a basic notion of topics. It was adapted to implement the privacy-preserving subscription list, based on Bloom filters. It also lacked any repair mechanism for the repair layer. To that end, it was also adapted to use the Reed-Solomon repair mechanism.

Even though the simulator implementation was mostly similar in terms of logic, the written code is largely different from a real-world implementation. This is though to optimizations that can be performed in the simulator but not in the real-world. For example, the simulator allows for shared data structures and variables. Furthermore, messages can be passed directly between nodes without the need for serialization. Hence, most of the time costs on the simulator are not calculated based on the actual size of the message but based on an approximation to the real value. These optimizations allow for a faster deployment using bigger systems and without consuming as many resources.

Chapter 5

Experimental Results

The experimental evaluation compromises both the real-world system and the simulation environment. We start by performing a small scale experiment between the real-world system and the simulator. This allowed us to confirm that real-world results were possible to obtain in the simulation. The obtained results were used to further calibrate and increase the faithfulness of the simulation environment. By confirming that in the same conditions both systems produce the same results, we can then proceed to perform the experiments tackling more realistic deployments, involving more nodes. Therefore, large scale deployments can be tested in simulation only, with a good level of fidelity.

The experiments were run in a machine with the following characteristics. CPU: Intel i5-6500 @ 3,2GHz @ 4 Cores. RAM: 16GB. OS: Microsoft Windows 10 Education v10.0.16299. Java: JRE-10.0.2.

There are important observations to be made on the real-world scenario experiments. Java memory allocation pool allocates too much unnecessary space when starting the JVM. Though to our limited resources, each node had to be started using the -Xmx flag. This flag allows specifying the maximum size, in bytes, of the memory allocation pool. The value used in the flag was tuned to the point each node keeps the minimum memory necessary, without losing performance, enabling us to reduce the memory used by each node by half.

In the simulator, traffic costs are a direct sum of what the message in the real-world would cost. For example; an int variable would be summed as 4 bytes; and a long variable as 8 bytes. Although, the use of Kryo serialization compresses all types of variables, even Java primitive types. This means *int* values can costs as low as 3 bytes and *long's* can cost 4 bytes. Another interesting propriety is that the OpenBitSet implementation uses long's to store bits. Java long's are stored as 64-bit integers. Meaning the bitset size is a multiple of 64 bits. To precisely terminate the size of a bit set, after it has been serialized

with Kryo, we conducted a set of quick experimental evaluations. The results shown in table 5.1 are for the most common size after serialization. Near empty or full bitsets are compressed to smaller sizes, but those cases are rare.

Furthermore, traffic accounting includes: the size of the TCP/IP headers for all messages exchanged; and the Java serialization header over the Kryo byte array serialization, which has a fixed size of 27 bytes. Additionally, the simulator accounts for TCP/IP header overhead related to the connection handshake. The overhead of the AKKA protocol was not included in the traffic accounting of either system. We decided to exclude AKKA of the costs as any other protocol, with cheaper communication primitives, could have been used to build the system. Although adjusted to match the real-world scenario, the simulation environment follows a more pessimistic traffic accounting. Not only it counts for handshake overhead, but it also calculates the full cost of some messages that might be optimized by Kryo in the real-world scenario.

Filter Size in Bits	256	512	1024	10K	16K	32K
Size in Bytes	32	64	128	1250	2000	4000
Serialized Size in Bytes	44	80	152	1403	2233	4445

Table 5.1: Filter byte size after serialization

5.1 Membership Evaluation

The performance and behavior of the membership layer are inherited to the system churn, the filter size, and the repair mechanism interval. The system churn can be divided into two parts. Node churn, associated with the frequency on node arrivals and departures, and filter churn, associated with the frequency in which filters are updated. For simplicity, nodes do not reenter the system and each node arrival is treated as a fresh start.

The membership layer evaluation focuses on the costs of maintaining an up-to-date view of the system.

5.1.1 System Comparison

The content of this subsection is dedicated to establishing a comparison between both systems. Parameters were adjusted to so they would perform under the same conditions. The average node arrival, in the real-world scenario, was determined by our machine capacity to start nodes (JVM instances). Therefore, the average arrival rate was of 6 nodes per minute, with a total of 100 nodes.

The evaluation measured the costs of each individual system component while starting the system, as shown in Figure 5.1. Endpoint and Filter columns represent both the request message and the replays messages. JoinReq represents the message the new node sends to its leader when it is ready to join the system. Broadcast represents the sum of tree broadcast messages propagated in the system. Repair column is related to the

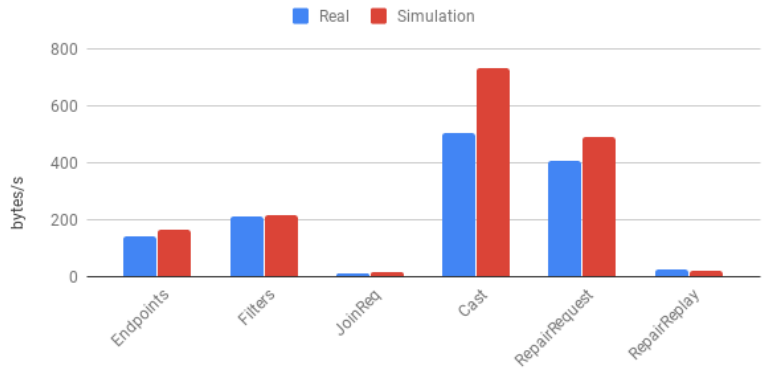


Figure 5.1: Costs of both systems when starting a 100 node system.

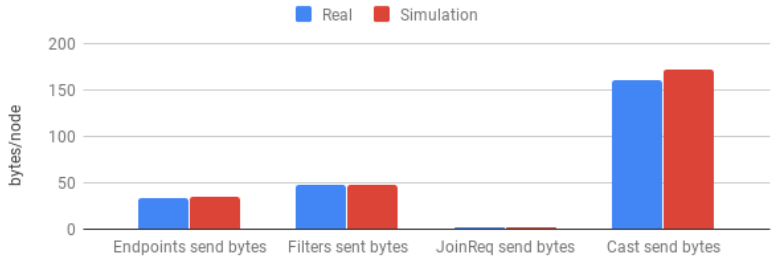
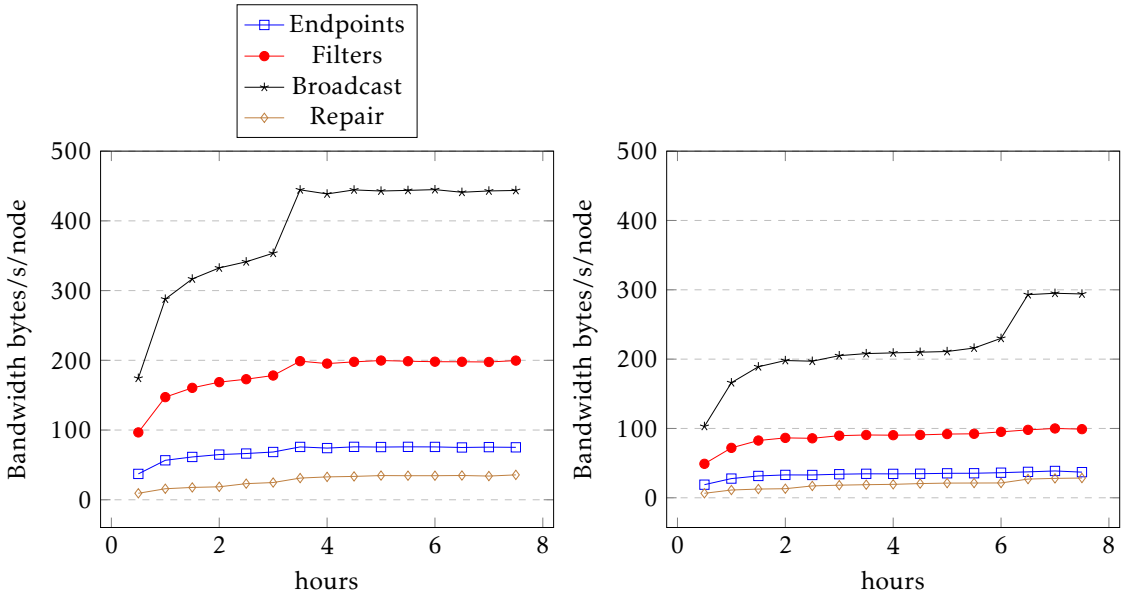


Figure 5.2: Costs of both systems for a new node arrival.

underlying recovery mechanism, while Repair Replay is when the recovery mechanism is successful in repairing the membership layer. There are some costs associated with this type of message due to the quick start ratio of the system. Occasionally, the repair mechanism repairs new tree broadcasts that are still being propagated throughout the system. Since the goals are to measure the membership costs, routing broadcast messages are not included.

The simulation environment accounts for traffic costs, such as the TCP/IP handshake, that were not accounted for in the real-world. Therefore, higher costs were expected in the simulation. This helps us prove that the simulator does not follow an overconfident solution that would prove unrealistic in a real-world solution.

A second experiment was conducted after the system had stabilized, with the goal of measuring the total cost of a single join. By analyzing the single cost of one join, it is easier to have a better perception of the system's cost. Repair costs are not shown since they are a result of the underlying repair mechanism and were not affect by the new node arrival. Figure 5.2 demonstrates that the major cost is to broadcast the new node arrival across the system. The cost to download both the endpoints and the filters can be divided by several participant nodes, distributing the load across the system.



(a) System A - high volume of churn.

(b) System B - lower volume of churn.

Figure 5.3: Membership bandwidth cost, per node, on two systems with the same number of nodes and different levels of churn. Bandwidth costs are mostly the same for Upload & Download.

	System A	System B
Weibull λ	2	2
Weibull κ	1.56	0.78
Session Duration	3-hours	6-hours
Avg new nodes per hour	8400	4200
Leader Aggregation Period	30 sec	30 sec
Repair Mechanism Interval	45 sec	45 sec
Fanout	2	2
Filter Size	512 bits	512 bits

Table 5.2: System parameters for first membership experimental evaluation. The goal of the experiment was to prove that the cost of maintaining the membership is associated with the level of node churn.

5.1.2 Membership Costs Evaluation

In a typical structured overlay, such as the one implemented, the majority of costs come from maintaining the membership, i.e., the cost of node churn and filter churn. We intend to demonstrate that systems with a higher ratio of churn are more expensive than systems with a lower rate of churn and that the system becomes inexpensive when stable.

The first experiment was conducted by using two systems with the similar configurations. While executing, both systems had an average of 25.000 nodes participating in the system. The systems differed in the average node arrival rate and the node session duration. The used parameters and arrival rates can be seen in Table 5.2. System **A** had a faster arrival rate and short session durations than System **B**, meaning that system **A** had a higher churn rate. The arrival rate resulting from the Weibull distribution was mixed with a random value, randomizing the average arrival rate to create periods with non-uniform arrivals. As the goal was to measure node churn, both the routing layer communication and filter updates were disabled.

Figure 5.3 shows the costs of both system, as they build-up over an 8-hour period. The number of nodes in system **A**, portraited on the left, stabilize at the around the 3,5h hour mark. On system **B**, portraited on the right, it only stabilizes at around the 6,5 hour mark. As both systems had an average of 25.000 nodes, new nodes had to download the same number of endpoints and filters when joining. The graphics demonstrate that system **A** has a higher cost, proving that the cost of the system is mainly related with node churn and not due to the system size.

If the system is stable, i.e., without constant joins, leaves or filter updates, the only associated cost will be regarding the repair mechanism. The overall repair cost is associated with the number of nodes in the system, the number of lost messages, and the size of the version vector being used. The version vector size is mostly dependent on the number of leaders that emitted timestamps. When the system is initial growing, the leader of a slice changes quickly due to new nodes arriving with lower keys. This causes the vector to increase in size as it stores entries of previous leaders. When the system progresses or previous leaders leave, it is possible to clean the vector of past timestamps.

Despite the high levels of node churn, shown in system **A**, the repair mechanism costs is akin to the costs in system **B**, which has half the node churn level. Hence, the cost to maintain the repair mechanism is minimal.

5.1.3 Costs on different scenarios

This subsection is focused on measuring the membership costs of the two scenarios introduced in Chapter 3.

System **X** portraits the large scale scenario, were participant nodes have a tendency to join and leave the network, causing churn. Nodes are the final destination of the events, so modest sized filters can be used. We chose filters with 512 bits, which can store around

	System X	System Y
Weibull λ	2	1
Weibull κ	0.78	0.47
Session Duration	6-hours	∞
Avg new nodes per hour	4200	Stable
System Size	25.000	1.000
Leader Aggregation Period	30 sec	30 sec
Repair Mechanism Interval	45 sec	90 sec
Fanout	2	2
Filter Size	512 bits	64Kbits
Filter Update Frequency	60-minutes	5-minutes

Table 5.3: System parameters for the two proposed solution scenarios.

50 topics with 1% FPP and 80 topics with 5% [1]. The system was modeled to have an average of 25.000 participant nodes, with session durations of 6 hours, and that update their filters every 60 minutes.

System **Y** portraits a scenario were participant nodes act as brokers for a large number of client nodes - the actual event subscribers. Broker nodes do not leave the system, so there is no node churn, and their number is limited to 1.000 nodes. Client nodes do not participate in the system, in particular, at the membership level, acting merely as clients. But they come and go rapidly, and due to their volatility, the filters managed by the broker nodes are prompt to change frequently, causing filter churn.

New client nodes might have a portion of the topics in common with client nodes already in the system. Hence, we assumed that brokers only need to emit filter updates every 5-minutes. If the broker does not have some of the interests from a new node, a 5-minute delay until the new client node starts to receive events might be too much. In those cases, since brokers know the filters of each other, the new client node could be temporarily redirected to a broker with its interests.

Brokers might be distributed geographically, so their associated external nodes might be interested in a widely different range of topics depending on the region. On account of that, and the sizable number of clients expected, broker filters need to be considerably large. We chose filters with 64Kbits, which can store around 6500 topics with 1% FPP and 10000 with 5%. Finally, the membership repair interval was increased on the base that the system is stable and message loss will be rare. The summary of the parameters used in both scenario can be seen in Table 5.3.

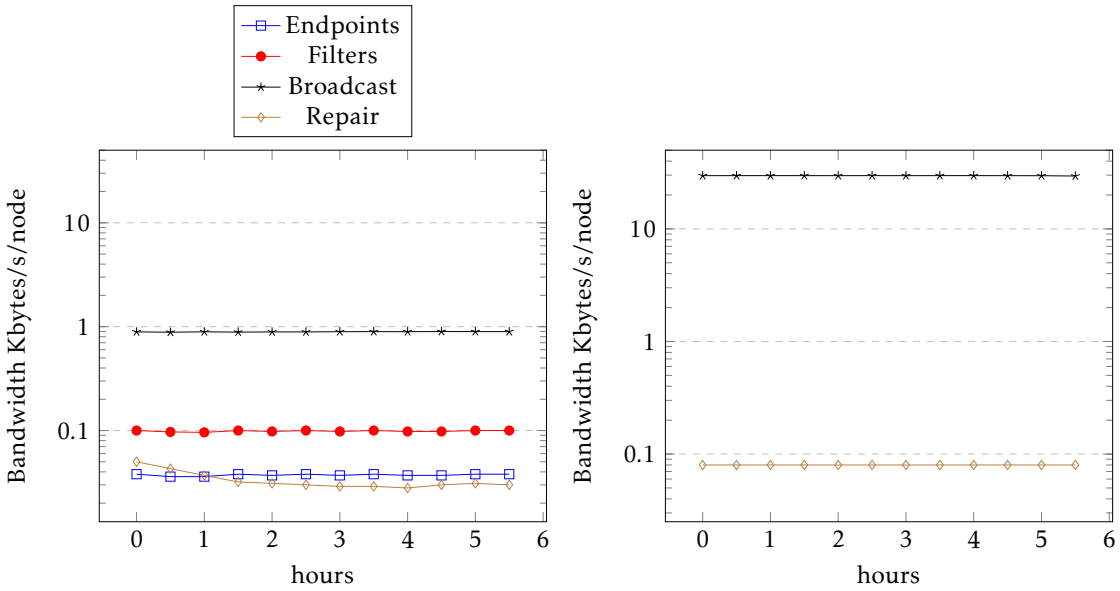
Figure 5.4 shows the result of the experiment over a 6 hour sample, showing system **X** on the left and system **Y** on the right. Since in system **X** nodes join and leave at the same time, the system keeps an almost constant cost during the experiment.

Constant node churn represents a significant portion of the total cost for the membership. New nodes need to obtain the system view (endpoint and filters) and their

[1] as seen before in Table 4.1.

information needs to be broadcasted to the remaining of the network. The cost to download the system view is amortized the longer the node session is, and most of the cost falls to the cost of broadcasting filter updates. Furthermore, the cost per node to maintain the membership, at around 1KBytes/s, is considerably inexpensive and can be easily supported by this type of system.

System **Y** has no node churn, therefore, there are no costs associated with endpoint or filter download. Repair costs are also minimal since the system is stable and reliable. The cost to maintain the membership, with such levels of filter churn, is much higher than in the previous example. Although, nodes in this scenario act as brokers and the scenario is akin to the 5G cell towers. Therefore, the cost of around 30 Kbytes/s becomes inexpensive for this scenario and can be easily supported by the broker nodes.



(a) System X - average size of 25k nodes.

(b) System Y - fixed size of 1k nodes.

Figure 5.4: Membership bandwidth cost for the proposed scenarios. Bandwidth costs are mostly the same for Upload & Download.

5.2 Routing Evaluation

This section is dedicated to evaluating the different components of the routing layer. We want to prove that the amount of work performed by a node is proportional to the number of topics it subscribes. The work ratio can be affected by the number of ON bits and the FPP of the filter. Hence, we also want to study how those components affect the work performed by nodes. Finally, we are going to determine the effectiveness of the error correction solution.

To simulate event topics, nodes use a text file containing 300.000 different words, which can be selected as topics. Topic popularity follows a real-world scenario where some topics are more popular than others. This is, topics can be popular and be subscribed by a big portion of the nodes in the system. Or unpopular, only being subscribed by a few nodes. The popularity levels were modeled using a Zipf's distribution [52] with an exponential parameter of 1, contained in the Apache Math Commons library [5].

5.2.1 System Comparison

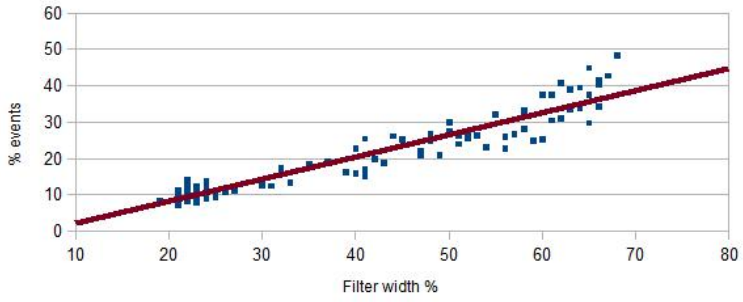


Figure 5.5: Real-world routing layer evaluation.

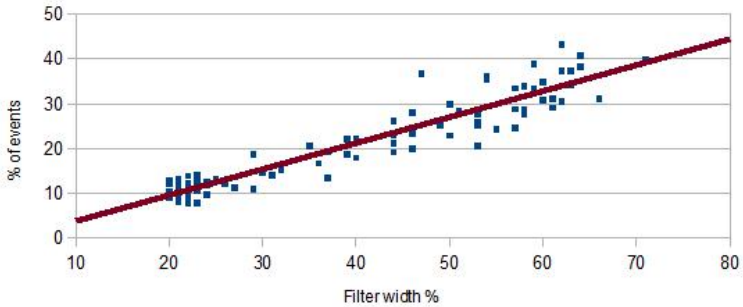


Figure 5.6: Simulation routing layer evaluation.

An initial experiment was performed to compare the behavior of both systems in the routing layer. To reduce any potential randomness that could affect the results, both were started with a list of pre-arranged filters. The topics contained in each published event

were also created using a random generator with the same seed. The experiment was conducted in a system with a stable number of nodes, i.e., new node arrivals or exits are non-existent. Therefore, event publishing begins after the membership layer has stopped receiving new nodes, and all the nodes have their views up-to-date. The experiment was conducted over a period of 1 hour and a total of 6.000 events were published in the system. It is the correspondent to each node publishing a new event every minute, during the full 1 hour execution time.

Figure 5.5 shows the results of the real-world implementation and Figure 5.6 the simulated environment. Each dot in the graphic represents a node in the system. The experiment compares the percentage of filter width occupied (% of ON bits) with the percentage of work performed. The work performed is expressed in the percentage of messages sent by a node compared to the total number of events published in the system. The results show that both systems have delivery the events in an almost exact same fashion. They also show that as the node subscribes to more topics, increasing the filter width percentage, it also performs more work.

5.2.2 Fair Resource Distribution

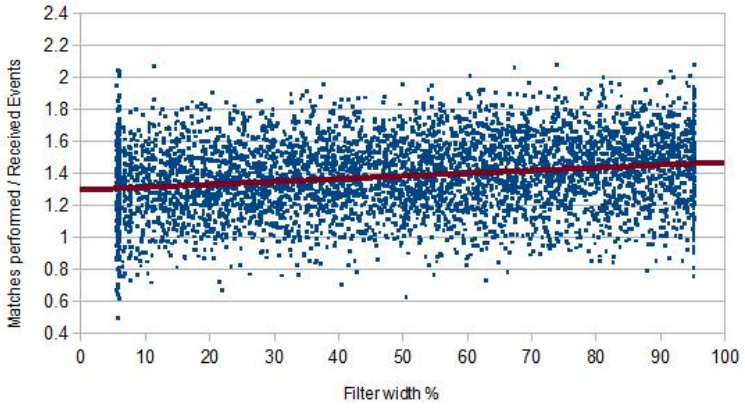


Figure 5.7: Work ratio with uniform filter distribution.

One important feature of the routing layer is that nodes should perform work proportional to the size of their interest list. To isolate the amount of work performed, we started the system using filters with low FPP. Thus, ensuring that routing is performed in filters that do not suffer from false positives, as false positives would prompt nodes to perform more work than expected. For this experiment, filters had a bit size of 1.000 bits, and false positives were kept under 0,1%. The experiment was conducted on a simulation environment with 5.000 nodes, with a total of 6.000 different routing events published in the system during the execution time. We concluded that for this type of experiment, 6.000 events was enough to provide representative results. The routing layer propagated events with a fanout of 3. Using a random uniform distribution, nodes subscribed to a

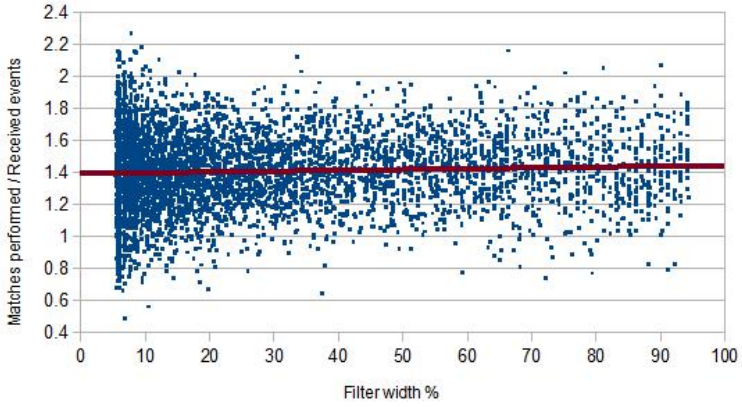


Figure 5.8: Work ratio with narrow filter distribution.

number of topics that made their percentage of ON bits range from 5% to 95% of the filter width.

Figure 5.7 shows the correlation between the ratio: Matches Performed / Events Received and the percentage of filter width occupied (% of ON bits). The value matches performed, represents the number of filter comparisons executed by the node until it finds an acceptable node to forward the event. It can be observed that the average work ratio stays within the same range across the full filter range, slightly increasing as the filter width reaches its maximum capacity. This indicates that nodes are keeping a work ratio proportional to their subscriptions list, i.e., as their subscription size grows, so does the number of matches performed.

In a real-world scenario, is not likely that the percentage of ON bits follows a uniform distribution. A more reasonable approach is that most nodes subscribe to a small number of topics. We carried an experiment to determine the correlation between the FPP and the percentage of ON bits, which can be seen in Annex II.1. This experiment aligned with the results seen before in Table 4.2, and showed that the Bloom filter achieves its peak performance when it is 50% full. Taking that into consideration, we modeled the experience using a popularity Zipf distribution of $\kappa = 0.75$. This gave us a distribution where most nodes use less than 50% of the filter. As shown in figure 5.8 the work ratio stays proportional to the number of subscriptions, even when the percentage of ON bits does not follow a uniform distribution. Annex II.2 shows a complementary experiment for when the distribution is mostly wide, that still maintains a balanced work ratio.

An additional remark can be made to the work ratio range. There seem to be nodes that tend to perform unbalanced amounts of work, when compared to the average. This condition is not caused by the size of the sample but rather by an unbalanced event dissemination tree. There are two characteristics of the tree division that must be taken into consideration. First, the key range space is not perfectly distributed. Second, the key range division takes into consideration the remaining number of nodes per slice, but there

are no guarantees that any of the remaining nodes subscribe to the desired topics. The combination of these characteristics causes small discrepancies in the amount of work performed by some nodes.

We designed a simple optimization to face the discrepancies, based on a dynamic routing fanout. If the node has been performing more work than that expected, it decreases the fanout. Otherwise, if it has performed less work, it increases it. Figure 5.9 and 5.10 shows the results using a dynamic fanout on a uniform and on a narrow distributed systems. The results demonstrate that a dynamic fanout can accommodate for the tree division imperfections, and provide the system with a better work ratio.

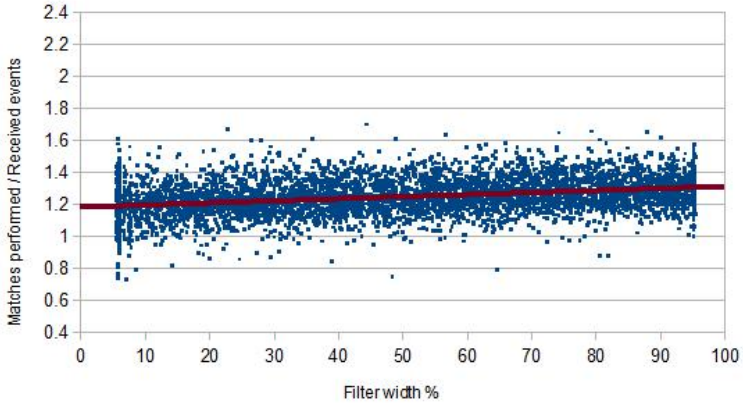


Figure 5.9: Work ratio with uniform filter distribution and dynamic fanout.

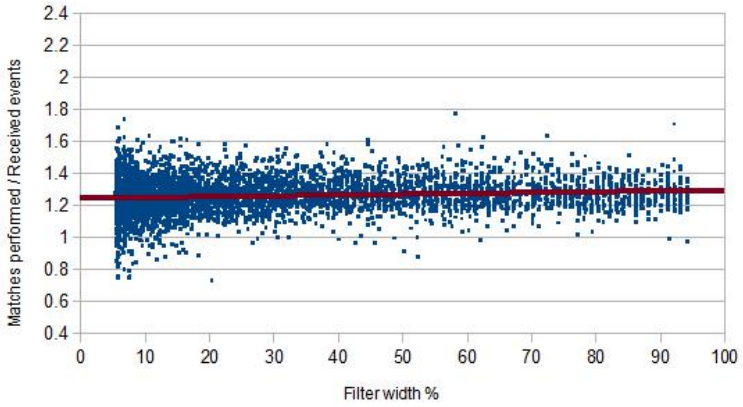


Figure 5.10: Work ratio with narrow filter distribution and dynamic fanout.

5.2.3 False Positives Impact

Another important experiment is related to the amount of work performed when there are false positives. This experiment was conducted by fixing the number of hash functions

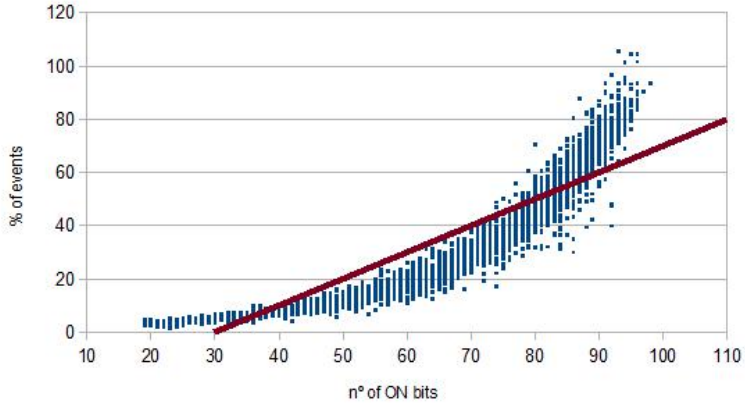
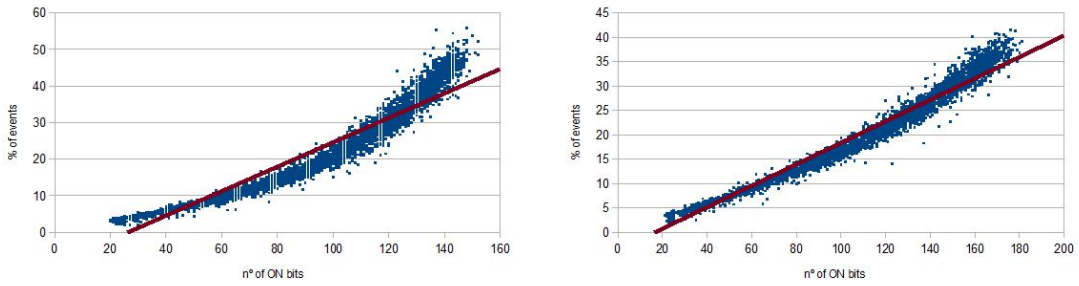


Figure 5.11: Load Distribution on 100 bit filter.

used in the Bloom filters, and by fixing a value for the max number of topics a node could subscribe to. Therefore, a node could subscribe to a random number of topics between 5 and the max number of topics, 50. We limited the minimum size by 5 in order to avoid nodes with subscriptions too narrow. The number of hash functions was set to 5. There is also a system parameter related to the number of topics available in the system. For this experiment, it was set to 3 times the max number of topics a node could subscribe, i.e., there is a total of available 150 topics in the system. A system with 5.000 nodes was used, running for a period of 4 hours. There was a total of 6.000 events published in the system and the routing fanout was set to 3.

Previous we evaluated the work ratio based on the number of matches performed divided by the number of events received. In these experiments, we want to see how the FPP affects the number of events a node receives. If we were to use the previous metrics, we would get work ratio charts similar to the ones seen before, as false positives would translate in more ON bits in the filter. That would not allow us to perceive how many more events a node was receiving. This section is evaluated based on the percentage of events a node receives versus the total number of events published in the system.



(a) 200 bit filter

(b) 300 bit filter

Figure 5.12: Load Distribution on 200 and 300 bit filters.

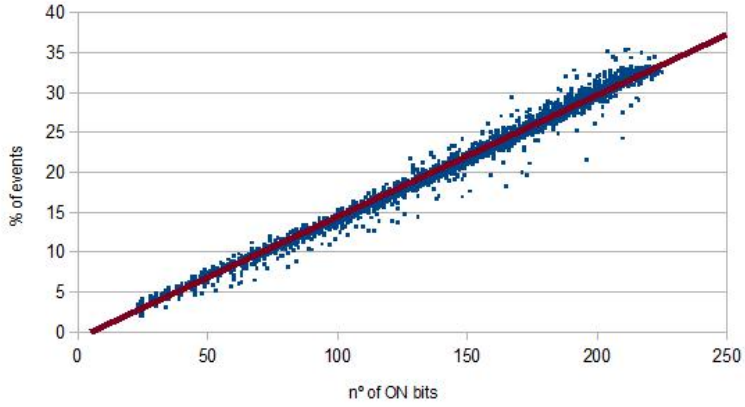


Figure 5.13: Load Distribution on 1000 bit filter.

Figure 5.11 shows the initial test conducted for a Bloom filter with 100 bits and the parameters described above. Figure 5.12 and 5.13 show the work distribution as we increase the filter size and keep the number of hash functions fixed. Additional graphics, showing a more gradual increase in the filter size, can be found in Annex II.1. In the first filter, with a size of 100 bits, some nodes received 100% of the published events. Using a filter with 1.000 bits, no node received more than 40% of the total published events.

The experiment proves that when the filters are too small, the limited number of bits causes several hash collisions. This affects the false positive rate, which affects the distribution of work in the system. By increasing the filter size, the FPP lowers, allowing nodes to receive events proportionally to the size of their subscription list.

As the filter grows, so does the amount of ON bits, reducing the false positive rate. Changes in the number of ON bits can be observed in filters up to 1.000 bits. At that point, the amount of false positives becomes inconsequential, as shown in Table 5.4. Although, starting from filters with 500 bits, the amount of FPP becomes almost insignificant. A quick reminder that the filter size has an impact on the membership costs, while the number of false positives in the routing layer. Thus, keeping the filter as small as possible with an acceptable amount of false positives is ideal for the performance of both layers.

Filter Size	100	200	300	400	500	600	700	1000
% of FP	42,6%	16,5%	5,26%	1,79%	0,57%	0,20%	0,30%	0,01%

Table 5.4: Correlation between the filter size and the amount of false positive messages sent in the system

5.2.4 Routing with Error Correction

This section presents experiments conducted using the Reed-Solomon strategy. The goal was to test the effectiveness of the solution in preventing the system from losing routing

events, within the expected failure rate. The base assumption is that in these types of systems the chance of a message being lost is quite low. The fail chances introduced in the experiments will translate in the probability of a node receiving an event and failing to continue its propagation. The aim is to determine the number of events that can still be decoded in the presence of failures.

We define the type of Reed-Solomon strategy used by defining the total number of shards and then the number of data shards. For example, a strategy with a total of 3 shards and 2 data shards, would be denominated T3-D2.

To test the solution, we used differently sized systems. Thought to the high number of event routing messages exchanged using the different Reed-Solomon strategies and to our limited time, the experiments where performed in a scaled down scenario. Therefore, the experiments where not tested in a scenarios with 25.000 and 5.000 participant nodes as in the membership evaluation, but with 5.000 and 500 nodes. The scenario with 5.000 nodes is denominated system **W** and the smaller scenario denominated system **Z**. The results were taken in stable systems, with no node or filter churn, and with a total of 6.000 published events. The routing fanout value was set to 2.

Figure 5.14 shows the difference between using a Reed-Solomon strategy T3-D2 and without using any recovery mechanism, for both scenarios. Without using any recovery mechanism, the total number of undecoded events quickly renders both systems unusable, even for a fail chance of 1%. This was expected, as with a low fanout, losing a message can mean losing up to half the event dissemination tree. The T3-D2 strategy still suffers from almost 3% of undecoded events with a fail chance of 1% for system **W**, and just under 2% for system **Z**. It is important to acknowledge that 1% is a high fail chance for both scenarios, so the results were expected.

The propagation trees in a system with the size of scenario **W** and fanout 2, need several hops to accommodate the full range of interested nodes. This means the chance of more than one shard being lost are considerable and explains why it performs worse than system **Z**.

	T3-D2		T6-D4		T9-D6	
Param Fail Chance %	W	Z	W	Z	W	Z
0.01	0.0006	0	0	0	0	0
0.05	0.0067	0.0037	0.0001	0	0	0
0.1	0.0662	0.011	0.0021	0.0003	0.0001	0

Table 5.5: Percentage of undecoded events using different Reed-Solomon strategies.

5.2.4.1 Reed-Solomon with multiple strategies

Figure 5.15 shows the percentage of undecoded events using different Reed-Solomon strategies. The percentage of undecoded events with small fail chances were hard to

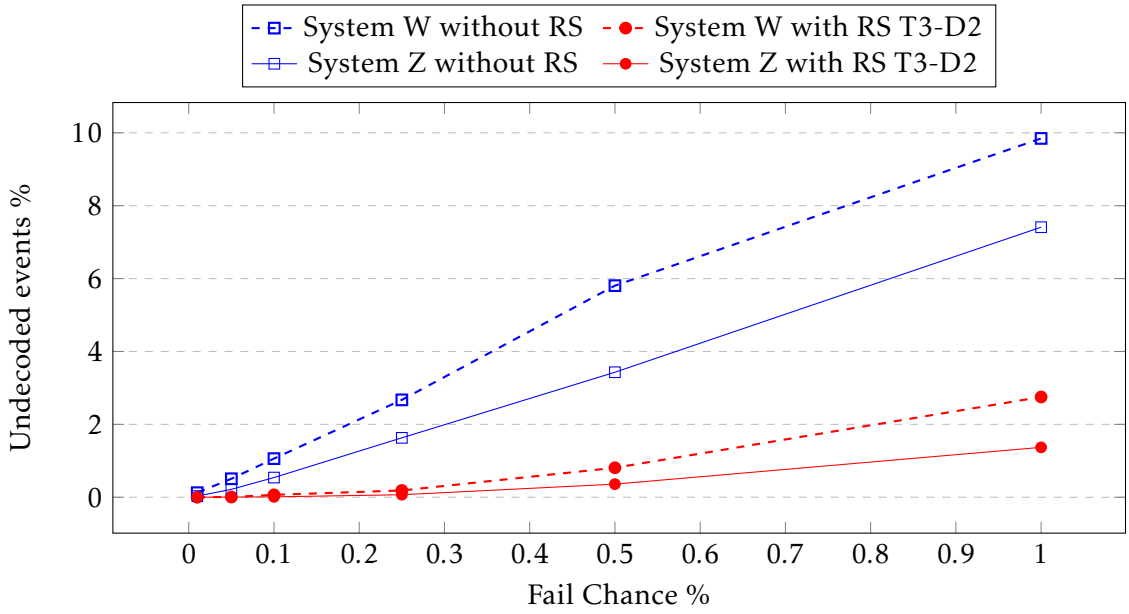


Figure 5.14: Percentage of undecoded events using a reed-solomon strategy T3-D2, and without using any recovery mechanism.

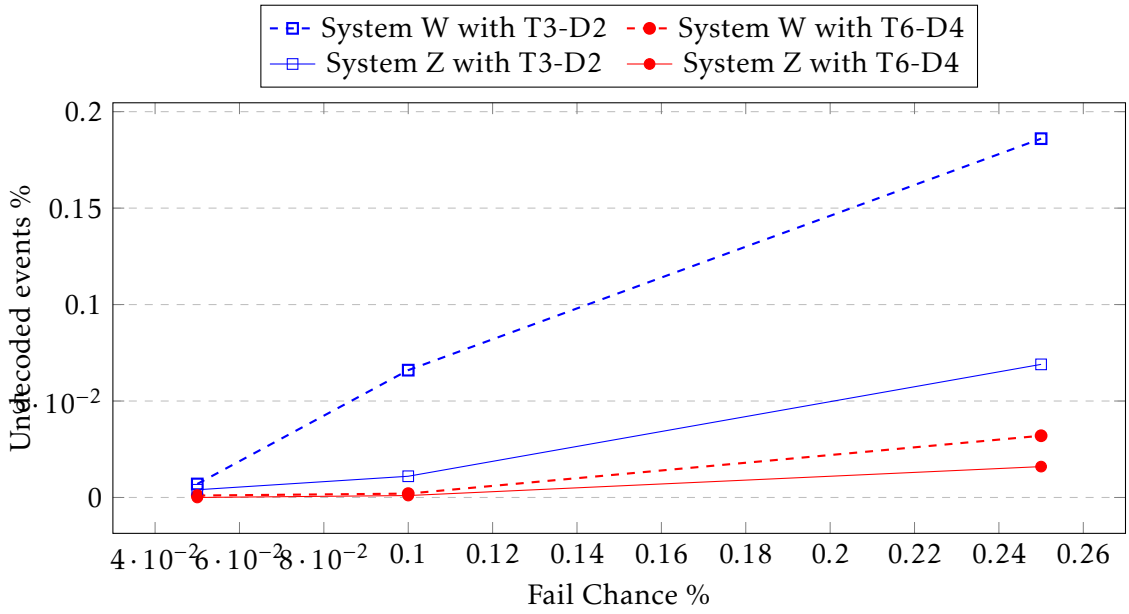


Figure 5.15: Percentage of undecoded events using different Reed-Solomon strategies.

represent in the same chart and therefore are shown in table 5.5. Each of the shown strategies has the same 3 to 2 proportion of needed shards, i.e., for every 3 shards there are 2 data shards. This roughly translates to having the ability to lose less than 34% of the messages and still being able to decode the message. Even though the proportion is the same, strategies with more shards have a better success probability.

Some of the used strategies were able to maintain the system with 0% of undecoded events, hence, confirming the effectiveness of the solution. Although, in a system running for a longer period of time with more published events, chances are, some events may remain undecoded.

In a third experiment, the total number of shards was fixed and the number of data shards changed through the experiment. Table 5.6 shows the results with the strategies: T6-D4, T6-D3, and T6-D2. As expected raising the number of parity shards improves the ability to decode events. The appropriate selection has to be made between a compromise in the tolerated number of failures and the size of each shard, as lowering the amount of shards needed to decode an event, increases the size of each shard.

	T6-D4		T6-D3		T6-D2	
Param Fail Chance %	W	Z	W	Z	W	Z
0.01	0	0	0	0	0	0
0.05	0.0001	0	0	0	0	0
0.1	0.0021	0.0003	0	0	0	0
0.25	0.032	0.016	0.0006	0	0.0001	0

Table 5.6: Percentage of undecoded events using different Reed-Solomon strategies, with the same total number of shards.

5.2.4.2 Fanout Affect on Reed-Solomon

One interesting behavior of the Reed-Solomon is how the same strategy behaves with different fanouts. When the fanout is small, it takes several hopes in order to reach the entire range of interested nodes. As the fanout increases, the number of hopes lowers, making the messages reach all interested nodes in fewer steps. It also means that each portion of the slice is divided into smaller slices quicker. As a result, if a message gets lost, changes are a smaller part of the range is affected. Therefore, bigger fanouts provide the system with a more reliable fail mechanism. Since the propagation tree grows exponentially with the fanout, different fanouts might output the same results, as they will cover the entire tree range in the same number of steps.

Figure 5.16 presents the experiments conducted with the same strategies and different fanouts on both scenarios. The experiments were conducted with a Reed-Solomon strategy of T3-D2, with a total of 6.000 published events. The experiment provided the necessary information to prove that bigger fanouts improve the reliability of the system.

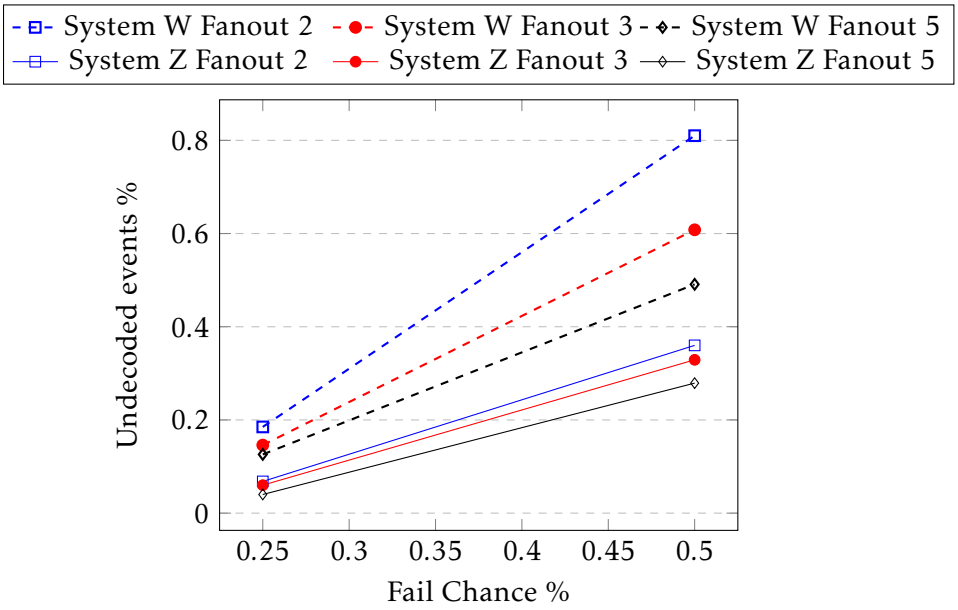


Figure 5.16: Percentage of undecoded events using the same Reed-Solomon strategy and different fanout values.

Since system **Z** is small, the different fanouts do not produce sizable improvements. Furthermore, for fail chance lower than 0.25% the benefits are minimal or indistinguishable from each other, even in experiments with more published events.

5.3 Conclusion

With the performed experimental evaluations, we proved that the real-world and the simulation environments produce comparable results. Hence, we were able to use the simulator to perform experiments in large scale scenarios and obtain trustworthy results.

By measuring the membership performance, in different scenarios, we determined that the cost to maintain this layer is related to the amount of node and filter churn, and not to the size of the system. The obtained costs proved that the membership is inexpensive and compatible with well-connected nodes working in the internet periphery, i.e., nodes in the proposed scenarios.

In the routing layer, the experiments proved that participant nodes perform a balanced amount of work. The process is fair and as the number of interests of a node increase, so does the amount of work it needs to perform. Finally, the Reed-Solomon strategy proved to be highly effective in reducing the number of lost events. Although, it is not enough to ensure that there are no false negatives in the system, as sporadically events might get lost.

Chapter 6

Conclusion

This chapter presents the conclusion of the thesis work and it also offers some possibilities of system improvement and some research opportunities.

6.1 Conclusion

The goal of this thesis was to develop a decentralized pub/sub solution. We proposed the solution for two different scenarios. The first scenario was characterized by having a large number of participant nodes, with decent bandwidth capacities. Though to their limited sessions, the scenario had high node churn, but low filter churn. On the second scenario, participant nodes acted as brokers for a larger number of client nodes - the actual event subscribers. Brokers had good connection capacities, although the client nodes where volatility. Hence, the scenario was portrayed by having no node churn, but high filter churn.

The system aimed to distribute work among the participant nodes, in a fair manner, independent of the scenario. Therefore, the amount of work performed by nodes was reflected in the size of their subscription list. Moreover, by creating random dissemination tree on the fly, participant nodes alternate between interior nodes and leaf nodes. Our goal was also to provide a privacy component, allowing participants to share their interests, without explicitly exposing them. Nodes supplied a list of topics they were interested in, in the form of Bloom filters. Since Bloom filters represent a subscription list in the form of an array of bits, the topics can be privately shared.

Through the experimental evaluation, we were able to prove that the real-world implementation and the simulation environment produce comparable results. It allowed us to use the simulator in large scale environments to produce trustworthy results. We proved that the cost to maintain the structured overlay is mainly caused by churn, and

not to the size of the system. In the routing layer, the experiments proved that the amount of work performed is distributed in a fair manner, i.e., nodes with wider subscriptions perform more work. By dynamically adjusting the fanout, according to the amount of work performed previously, we were able to improve the work distribution even further. Experimental evaluation of the filter size allowed to find a balance between the number of false positives and excessive filter size. Finally, we proved that the Reed-Solomon solution is an effective way to reduce the number of lost events. Although, it is not enough to ensure that there are no false negatives in the system.

The developed work provided several contributions:

- The model for a decentralized Publish/Subscribe aimed for the edge of the main Internet. The system distributed event routing work in a fair manner, without compromising the privacy of the participants, keeping the topics and event payloads concealed from the rest of the system.
- A real-world prototype of the Publish/Subscribe model described.
- A valid and extensive experimental evaluation, that attested the goals of our system through the use of the simulated environment and real-world environment.

6.2 Future Work

In the introduction, we briefly hinted that the performance of the system is inherently tied to the topology of the overlay. Despite, a topology based on neighborhood proximity or similar subscription interests was not developed as part of our approach. Thought to the lack of filter update, in nodes in the first described scenario, the topology could be organized based on similar subscription interests. Organizing the topology based on neighborhood proximity seems like a wise solution for the scenario where nodes act as brokers since this scenario is akin to the one used in 5G cell towers.

One focus of the thesis was related to the cost of maintaining a membership layer were nodes had a view of the entire system. Even though the key space is randomly rotated on every broadcast, we noticed nodes tend to only contact other nodes in front of them. This happens since the key range progression always follows an incremental order, until it reaches the maximum key and then starts over from the minimum key. Nodes might not need to have a view of the entire system, but only a portion in front of their key. There is a possibility to study a system where nodes only need to keep a partial view of the system, which would decrease the cost to maintain the membership.

The use of Reed-Solomon proved to be an effective method to prevent event loss, although it can sporadically lose events. Furthermore, the cost to keep such mechanism is high. As hinted before, we could use the filter information provided by the membership layer and compute nodes with a similar range of interests. This information could be

used to provide an epidemic repair mechanism akin to the membership layer but using bitwise similarities between filters.

Bibliography

[1] Akka. *Akka*. URL: https://akka.io/. Last visited on 2019/01/12.

[2] *Akka Serialization*. URL: https://doc.akka.io/docs/akka/2.5/serialization.html. Last visited on 2019/01/12.

[3] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang. "What will 5G be?" In: *IEEE Journal on selected areas in communications* 32.6 (2014), pp. 1065–1082.

[4] Apache. *OpenBitSet*. URL: https://lucene.apache.org/core/3_0_3/api/core/org/apache/lucene/util/OpenBitSet.html. Last visited on 2019/01/13.

[5] *Apache Match Commons*. URL: http://commons.apache.org/proper/commons-math/javadocs/api-3.4/org/apache/commons/math3/distribution/ZipfDistribution.html. Last visited on 2019/02/18.

[6] Backblaze. *JavaReedSolomon*. URL: https://github.com/Backblaze/JavaReedSolomon. Last visited on 2019/01/15.

[7] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni. "TERA: topic-based event routing for peer-to-peer architectures." In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 2–13.

[8] BitTorrent.org. *BitTorrent Specification*. URL: http://www.bittorrent.org/beps/bep_0052.html. Last visited on 2019/03/22.

[9] J. Blustein and A. El-Maazawi. "Bloom filters. a tutorial, analysis, and survey." In: *Technical Report CS-2002–10. Faculty of Computer Science, Dalhousie University* (2002).

[10] F. Cao and J. P. Singh. "Efficient event routing in content-based publish-subscribe service networks." In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. Vol. 2. IEEE. 2004, pp. 929–940.

[11] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure." In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.

[12] M. Castro, M. Costa, and A. Rowstron. *Peer-to-peer overlays: structured, unstructured*. Tech. rep. or both, 2004.

[13] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication." In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 14–25.

[14] K. Dolui and S. K. Datta. "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing." In: *Global Internet of Things Summit (GIoTS), 2017*. IEEE. 2017, pp. 1–6.

[15] B. Dupras. *Java-Cuckoo Filter*. URL: https://github.com/bdupras/guava-probably. Last visited on 2019/01/13.

[16] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and M. R. Torres. "2-3 cuckoo filters for faster triangle listing and set intersection." In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2017, pp. 247–260.

[17] EU. *General Data Protection Regulation*. URL: https://gdpr-info.eu/. Last visited on 2018/05/30.

[18] EU. *General Data Protection Regulation Information*. URL: https://www.eugdpr.org/. Last visited on 2018/06/06.

[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. "Cuckoo filter: Practically better than bloom." In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 75–88.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary cache: a scalable wide-area web cache sharing protocol." In: *IEEE/ACM transactions on networking* 8.3 (2000), pp. 281–293.

[21] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "Scamp: Peer-to-peer lightweight membership service for large-scale group communication." In: *International Workshop on Networked Group Communication*. Springer. 2001, pp. 44–55.

[22] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. "Meghdoot: content-based publish/subscribe over P2P networks." In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2004, pp. 254–273.

[23] A. Gupta. "Two hop lookups for large scale peer-to-peer overlays." In: *Proc. IRIS Student Workshop 2003*. Citeseer. 2003.

[24] A. Gupta, B. Liskov, R. Rodrigues, et al. "One Hop Lookups for Peer-to-Peer Overlays." In: *HotOS*. 2003, pp. 7–12.

[25] M. Hilbert. "How much information is there in the "information society"?" In: *Significance* 9.4 (2012), pp. 8–12.

[26] E. Lavoie, M. Correia, and L. Hendren. "Xor-overlay Topology Management Beyond Kademlia." In: *Self-Adaptive and Self-Organizing Systems (SASO), 2017 IEEE 11th International Conference on*. IEEE. 2017, pp. 51–60.

[27] D. P. Lawrey. *Avoiding Java Serialization to increase performance*. URL: https://dzone.com/articles/avoiding-java-serialization. Last visited on 2019/01/12.

[28] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees." In: *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE. 2007, pp. 301–310.

[29] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast." In: *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE. 2007, pp. 419–429.

[30] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. "A survey and comparison of peer-to-peer overlay network schemes." In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pp. 72–93.

[31] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric." In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[32] I. T. Neward. *Java Serialization 101*. URL: https://www.ibm.com/developerworks/library/j-5things1/index.html. Last visited on 2019/01/12.

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A scalable content-addressable network*. Vol. 31. 4. ACM, 2001.

[34] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, et al. "Handling churn in a DHT." In: *Proceedings of the USENIX Annual Technical Conference*. Vol. 6. Boston, MA, USA. 2004, pp. 127–140.

[35] L. Rizzo. "Effective erasure codes for reliable computer communication protocols." In: *ACM SIGCOMM computer communication review* 27.2 (1997), pp. 24–36.

[36] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems." In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[37] S. Setia. *Distributed Hash Tables (DHTs) Tapestry Pastry*. URL: https://cs.gmu.edu/~setia/cs699/lecture2.pdf. Last visited on 2018/05/13.

[38] M. Skjegstad. *Java-BloomFilter*. URL: https://github.com/MagnusS/Java-BloomFilter/blob/master/src/com/skjegstad/utils/BloomFilter.java. Last visited on 2019/01/13.

[39] E. Software. *Kyro*. URL: https://github.com/EsotericSoftware/kryo. Last visited on 2019/01/12.

[40] R. Staff. *Cambridge Analytica CEO claims influence on U.S. election*. URL: https://www.reuters.com/article/us-facebook-cambridge-analytica/cambridge-analytica-ceo-claims-influence-on-u-s-election-facebook-questioned-idUSKBN1GW1SG. Last visited on 2018/06/06.

[41] Statista. *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)*. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. Last visited on 2018/06/06.

[42] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications." In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

[43] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. "A peer-to-peer approach to content-based publish/subscribe." In: *Proceedings of the 2nd international workshop on Distributed event-based systems*. ACM. 2003, pp. 1–8.

[44] K. Thein. "Apache kafka: Next generation distributed messaging system." In: (2014).

[45] S. Voulgaris, D. Gavidia, and M. Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays." In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217.

[46] Z. Wang, G. Dong, and J. Zhu. "Dual-Kad: Kademlia-Based Query Processing Strategies for P2P Data Integration." In: *Web Information Systems and Applications Conference (WISA), 2012 Ninth*. IEEE. 2012, pp. 155–158.

[47] Wiki. *Bloom Filer*. URL: https://en.wikipedia.org/wiki/Bloom_filter. Last visited on 2018/06/05.

[48] Wikipedia. *Akka*. URL: https://en.wikipedia.org/wiki/Akka_(toolkit). Last visited on 2019/01/12.

[49] Wikipedia. *Error Correction Code*. URL: https://en.wikipedia.org/wiki/Error_correction_code. Last visited on 2019/02/28.

[50] Wikipedia. *Reed-Solomon*. URL: https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction. Last visited on 2019/01/21.

[51] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, et al. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing." In: (2001).

[52] *Zipf Distribution*. URL: https://en.wikipedia.org/wiki/Zipf%27s_law. Last visited on 2019/02/18.

ANNEX I

Annex Filter Benchmark

I.1 Filters Benchmarks

This additional section is dedicated to comparing the two distinct solutions of probabilistic sets seen before. Both are a type of data structure that minimizes the amount of memory used on queries, with the throwback of possible false positives.

The section is divided into the following way. First, we individualy study Bloom Filters, then the Cuckoo Filters. In the end, we make an in-depth comparison between the two filters, and draw some final conclusions about the more appropriate filter.

I.1.1 Bloom Filter

As studied before the Bloom Filter uses different hash functions to map an element to positions on the array. The rate of false positives can be decreased by increasing the number of hash transforms, up to a point, or increasing the array size. Therefore, we want to study how false positives affects the size of bit array and number of hash functions and what happens to the FPP as we insert more items then the supposed ones.

In the first experiment, we fixed the expected number of insertions in the filter. Then the desired FPP value was changed with the purpose of observing how it affects the size of the Bloom Filter. Graphic I.1 shows the result of the experiment for a fixed value of 100 and 1k insertions, as well as the expected size according to the theoretical formula. It is possible to notice, that in theory, the array size should be smaller than the one created by the application. This happens since the implementation doesn't actually create a Bloom filter with the exact FPP desired, but with an FPP smaller. More explicitly, the array with 100 insertions and 10% false positives, actually has an FPP of 6,3%, meaning the filter will have more bits per item. The size of the array grows on the same proportions for both tests, confirming that in order to decrease the FPP we need to increase the array size.

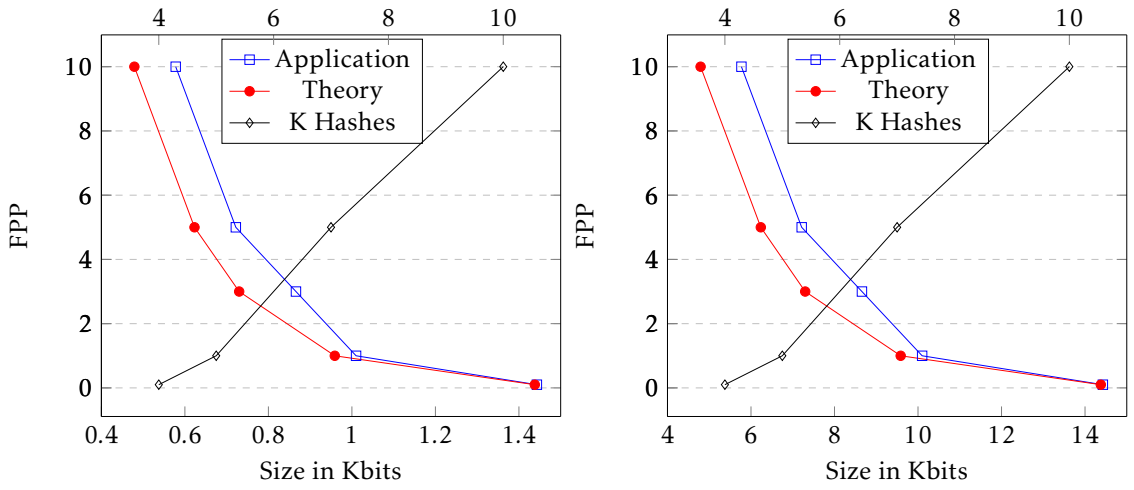


Figure I.1: Left: Bloom Filter size for 100 fixed insertions and different FPP. Right: Bloom Filter size for 1k fixed insertions and different FPP.

In the same graphic, it is also possible to see the number of hash functions used according to the FPP and the filter size. It shows that the number of hash functions, used by the implementation, only change according to the FPP and not with the number of expected insertions.

Left graphic I.2 presents the correlation between the initial FPP the filter was designed to support, and how that percentage changes if we insert more items then those expected. We also present a test denominated PFPP (practical FPP) in order to compare the theoretical formula with real-world application of the filter. This test takes a set of items two times bigger than the initial insertion, where none of the items is present in the filter. It builds the PFPP rate based on how many times the filter confirms the presence of a wrong item.

In the experiment, the initial filter was designed to support 100 insertions. Tests were conducted with an initial FPP of 5% and another of 10% FPP. It is possible to see that real-world application of the filter is practically identical to the theoretic formula. This also confirms a theoretical property of the bloom filter: the FPP is not affected by the size of the population but by the number of inserted items in the filter. The number of extra insertions shown in left I.2 was of: 5%, 10%, 30% and 50% more than the initially expected number of insertions. We conclude that if we designed a filter for a fixed number of elements, it will take up to 20% of the expected filters to match the initial FPP we desired. Beyond that it will still be possible to add more elements, compromising on the FPP.

I.1.2 Cuckoo Filter

The Cuckoo filter is a variant of the cuckoo hashing that stores only fingerprints. Fingerprint sizes are determined by the desired false positive rate, meaning smaller rates requires longer fingerprints to allow for fewer false positives. This filter can extend the

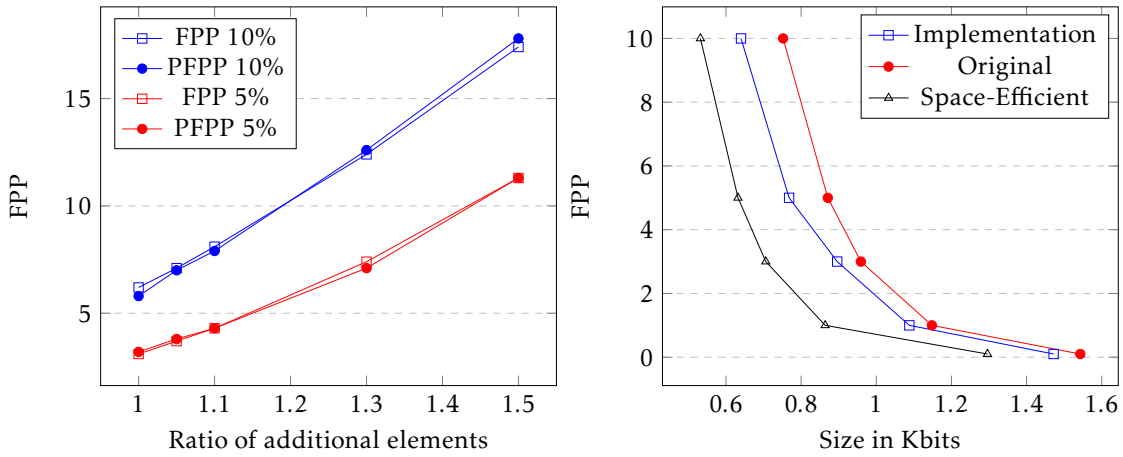


Figure I.2: Left: Inserting more elements then expected on Bloom Filter. Right: Cuckoo Filter size with 10k fixed insertions and different FPP.

amount of data that can be stored by allowing buckets to hold multiple items, leading to a tablespace that can be filled to 95% with high probability. Larger buckets improve table occupancy but require longer fingerprints to retain the same FPP. According to the theory, the number of bits per entry could be calculated using the following formula: $(\log_2(1/FPP) + 3)/\alpha$. Although the implementation tested [15] calculates the number of bits per entry using the following space efficient formula, also present in the paper: $\log_2(2b/FPP)$.

Right graphic I.2 shows the result of experiments made on the size of the array, as the FPP was reduced. The experiment was conducted measuring the bits returned by: the implementation, the original formula, and the space-efficient formula. As the reader can easily spot, there is quite a difference in bits between the space-efficient formula and the implementation. The gap can be justified since there must be some slack in table **??**. As a result, each item effectively costs more than a fingerprint size. Moreover, opposed to the Bloom Filter, the Cuckoo Filter desired FPP matches the implementation value.

Left graphic I.3 shows the correlation between the load factor of the cuckoo filter, its FPP and PFPP. The filter was created to hold 100 elements. According to the experiments, when a filter holds as many insertions as expected it has a load factor of approximately α = 84%. In the experiment, we tried to insert more 5% and 10% than the initial expected number of insertions. Although, beyond a load factor of α = 84% the filter becomes too saturated and refuses to accept more elements. Furthermore, there is a difference between the filter FPP and the RFPP caused by the small filter size.

I.1.3 Comparison and Conclusion

The premise for comparing both filters is that Cuckoo Filters are more efficient and use less space than Bloom filters when the false positive rate is kept under 3%. We had to take into consideration that the Bloom Filter implementation doesn't translate to the

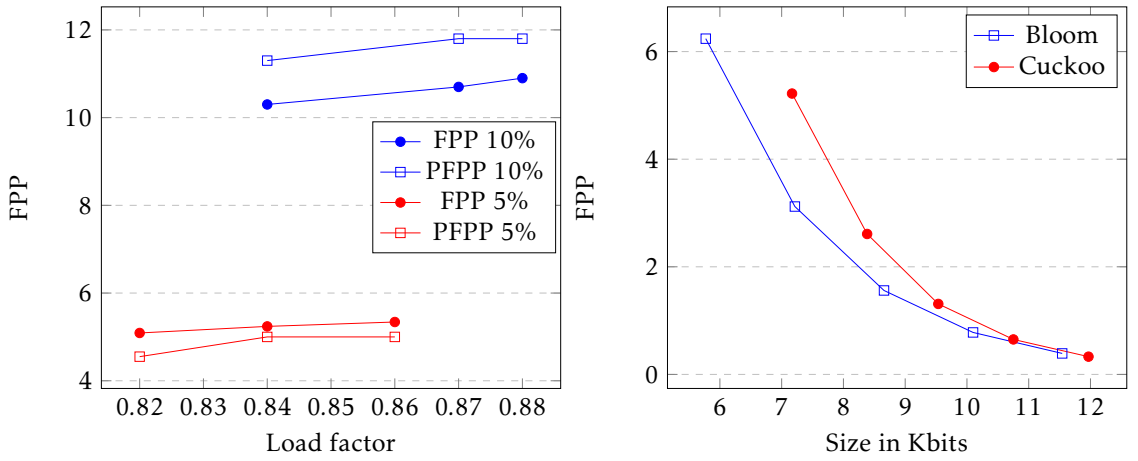


Figure I.3: Left: Cuckoo Filter size with 100 fixed insertions and different FPP. Right: Comparison between the two filters.

desired false positive value. That makes the Bloom Filter bigger since the FPP is smaller than desired. Therefore, we conducted an experiment were Cuckoo filters were initiated with the resulting FPP obtained from the Bloom filters. This allowed for a more accurate comparison between the two. Although, the Cuckoo implementation also slightly rounds the FPP value which created a small discrepancy between FPP.

Right graphic I.3 shows the result of the experiment. Theoretically, the Cuckoo Filters should be more efficient than Bloom Filter, at around 3% FPP. Although, the turning point was only observed at percentages lower then 0.5%.

Bigger filters mean a heavier cost for the membership layer. So reducing the filter size is beneficial, even at the cost of a small number of false positives. According to our experiments, Bloom Filters were constantly smaller than Cuckoo within an acceptable FPP range. Therefore, our choice for the more appropriate filter went to the Bloom Filter.

ANNEX II

ANNEX

II.1 Routing Evaluation Annex

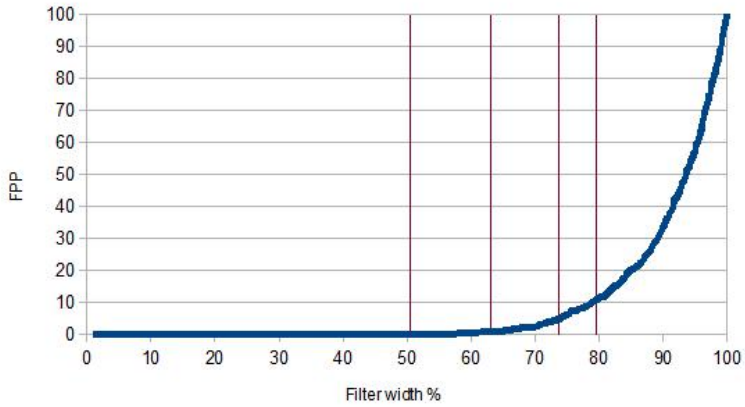


Figure II.1: Correlation between filter width percentage (% of ON bits) and FPP.

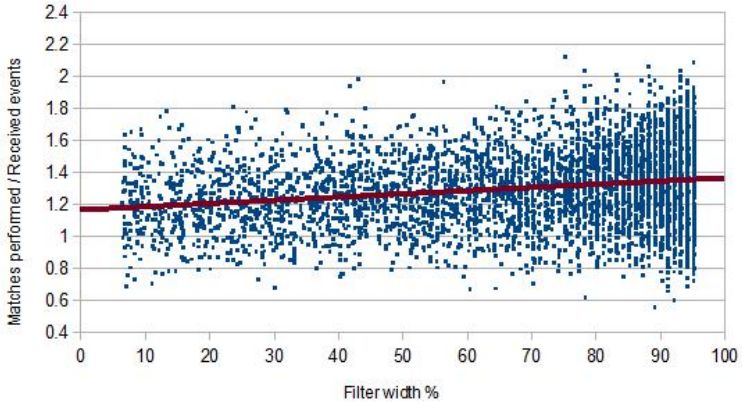
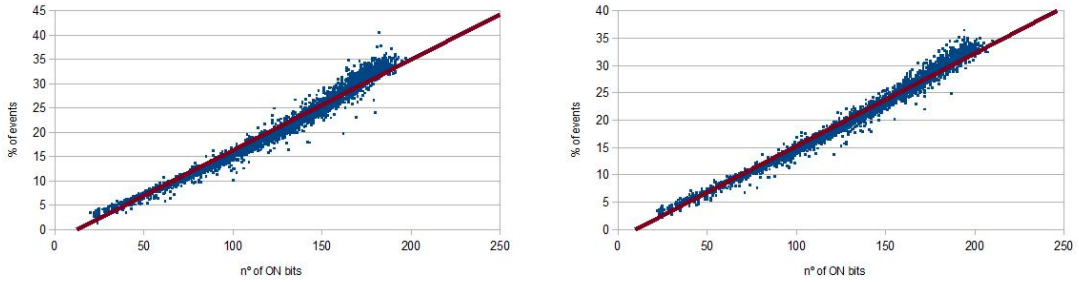
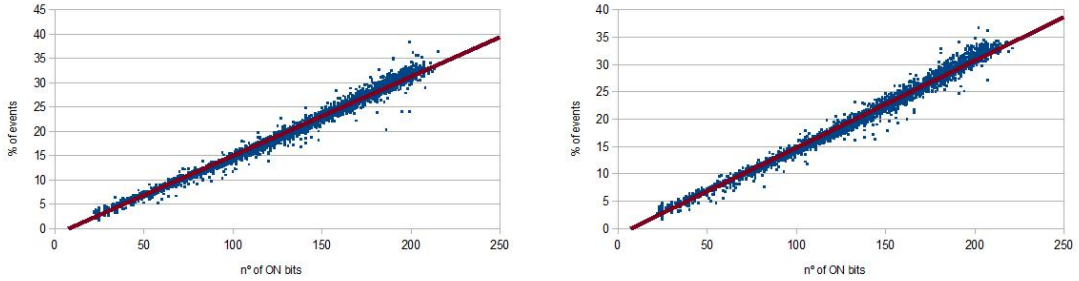


Figure II.2: Work ratio with wide filter distribution



(a) 400 bit filter (b) 500 bit filter

Figure II.3: Load Distribution on 400 and 500 bit filters.



(a) 600 bit filter (b) 700 bit filter

Figure II.4: Load Distribution on 600 and 700 bit filters.

II.2 Reed-Solomon Tables Annex

Needed Shards	Individual Shard Size	Sum of Shards	Total Transmission Cost
1/5	100	500	1000
2/5	50	250	750
3/5	34	170	670
4/5	25	125	625
5/5	20	100	600

Table II.1: Reed-Solomon reconstruction for 100 byte message using 5 shards in total.

Needed Shards	Individual Shard Size	Sum of Shards	Total Transmission Cost
1/6	100	600	1200
2/6	50	300	900
3/6	34	204	804
4/6	25	150	750
5/6	20	120	720
6/6	17	102	702

Table II.2: Reed-Solomon reconstruction for 100 byte message using 6 shards in total.

Needed Shards	Individual Shard Size	Sum of Shards	Total Transmission Cost
1/9	100	900	1800
2/9	50	450	1350
3/9	34	306	1206
4/9	25	225	1125
5/9	20	180	1080
6/9	17	153	1053
7/9	15	135	1035
8/9	13	117	1017
9/9	12	108	1008

Table II.3: Reed-Solomon reconstruction for 100 byte message using 9 shards in total.